FACHHOCHSCHULE **TRIER**
Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences
Informatik

Eine Webbasierte Entwicklungsumgebung für verschiedene Programmiersprachen

A web based Development Environment

Henning Voß

Betreuer: Prof. Andreas Künkler

Trier, 22.2.2007

Inhaltsverzeichnis

1 Einleitung....5
2 Ziele....6
3 Anforderungen und Konzept....7
3.1 Allgemeine Funktionen....9
3.2 Integration in Ilias....9
3.3 Compiler und Linker....10
3.4 Debugger....10
3.5 Sicherheitskonzept....10
3.6 MJVM....11
3.7 Zusammenfassung....12
4 Entwurf....13
4.1 Online-Compiler....13
4.1.1 Servlet....14
4.1.1.1 Dateiverwaltung....15
4.1.1.2 Programmiersprachen....17
4.1.1.3 Jobs: Compilieren, Ausführen und Debuggen....18
4.1.1.3.1 Linker....22
4.1.1.4 Nutzerverwaltung....23
4.1.1.5 Servlet-Infrastruktur....25
4.1.1.5.1 Kommunikationsprotokoll....28
4.1.2 Webinterface (Client)....31
4.1.2.1 Bestandteile der Benutzerschnittstelle....31
4.1.2.2 Funktionen der Benutzerschnittstelle....33
4.1.2.3 Layout....34
4.1.2.3.1 Dialoge....38
4.2 Multi-Task-Java....40
5 Implementierung....42
5.1 Online-Compiler....43
5.1.1 Das Servlet....43
5.1.1.1 Datei-Übersicht....43
5.1.1.2 Compilierung des Servlets....44
5.1.1.3 Implementierung des Servlets....45
5.1.1.3.1 Dateien....45
5.1.1.3.2 Ilias-Anbindung....46
5.1.1.3.3 C und C++....48
5.1.1.3.3.1 Compilieren....48
5.1.1.3.3.2 Ausführen....50
5.1.1.3.3.3 Debuggen....50
5.1.1.3.4 Java....55
5.1.1.3.4.1 Compilieren....55
5.1.1.3.4.2 Ausführen....56
5.1.1.3.4.3 Debuggen....57
5.1.1.3.5 Lisp und C#....60

5.1.1.3.6 AppArmor und Java-Security-Manager....61
5.1.1.3.7 User und Client-Sessions....64
5.1.1.3.7.1 Ereignisse....65
5.1.1.3.7.2 Session Synchronisation....66
5.1.1.3.8 Kommunikation Client-Servlet....69
5.1.1.3.8.1 XML-Antworten des Servlets....70
5.1.1.3.8.2 Kommunikationsprotokoll....71
5.1.1.3.8.3 Befehlsübersicht....72
5.1.1.3.9 Log-Datei....77
5.1.2 Das Webinterface....78
5.1.2.1 Datei-Übersicht....78
5.1.2.2 Implementierung der Benutzerschnittstelle....79
5.1.2.2.1 Dynamisches HTML....79
5.1.2.2.2 HTML-Code „einpflanzen“....81
5.1.2.2.3 Ereignisbehandlung....82
5.1.2.2.4 Quelltext-Editor....85
5.1.2.2.5 Übersicht Grafische Komponenten....88
5.1.2.2.6 Integration in Ilias....89
5.1.2.2.7 Kommunikation mit dem Servlet / MVC....91
5.2 Multi-Task-Java....94
5.2.1 Datei-Übersicht....94
5.2.2 Compilierung der MJVM....94
5.2.3 Demo-Programm ausführen....95
5.2.4 Implementierung der MJVM....96
5.2.4.1 Kommunikation zwischen Client-Anwendungen und einer MJVM....96
5.2.4.1.1 Verbindung zu einer MJVM herstellen....97
5.2.4.2 Ausführen von Java-Anwendungen....99
5.2.4.3 Die Klasse MJVMVirtualMachine....103
5.2.4.3.1 System-Klassen....104
5.2.4.4 MJVM-Client Klassen....106
5.2.4.4.1 Eine MJVM starten....107
6 Test....108
6.1 Einleitung....108
6.2 Systemtest....108
6.2.1 Testplan....108
6.2.2 Testspezifikation....109
6.2.2.1 Testfall 1....109
6.2.2.2 Testfall 2....111
6.2.2.3 Testfall 3....112
6.2.2.4 Testfall 4....114
6.2.2.5 Testfall 5....114
6.2.2.6 Testfall 6....116
6.2.2.7 Testfall 7....117
6.2.3 Testbericht....118
6.2.3.1 gefundene Fehler....118
6.3 Browsertest....119

6.3.1 Firefox....119
6.3.2 Internet-Explorer....119
6.3.3 Opera....120
6.3.4 Safari....120
6.3.5 Konqueror....120
6.3.6 Fazit....120
6.4 Last- und Servertest....121
6.4.1 Testplan....121
6.4.2 Testspezifikation....122
6.4.2.1 Testfall 1....122
6.4.2.2 Testfälle 2, 3 und 4....122
6.4.2.3 Testfälle 5, 6 und 7....122
6.4.2.4 Testfall 8 (MJVM)....123
6.4.3 Testbericht....124
6.4.3.1 gefundene Fehler....124
6.4.3.2 weitere Probleme....124
6.5 Test-Ergebnis....125
7 Server-Sicherheit....126
7.1 Tomcat....126
7.2 AppArmor....126
7.3 Java-Security-Manager....127
7.4 Zusammenfassung Sicherheitsvorkehrungen....127
8 Zusammenfassung....128
8.1 Ausblick....128
9 Anhang....130
9.1 CD-Inhalt....130
9.2 Administratorhandbuch....131
9.2.1 Installation des Online-Compilers....131
9.2.2 Konfiguration des Online-Compilers....133
9.2.3 Neustart des Online-Compilers....136
9.3 Benutzerhandbuch....137
9.3.1 Voraussetzungen....137
9.3.2 Das Hauptfenster....138
9.3.3 Menüpunkte und Funktionen....140
9.3.4 Dateien und Verzeichnisse....141
9.3.5 Programm-Beispiele....142
9.3.6 Dateien bearbeiten....143
9.3.7 Dateien lokal sichern....147
9.3.8 Compilieren von Dateien....148
9.3.9 Programme ausführen....149
9.3.10 Programme debuggen....150
9.3.11 Einschränkungen....152
10 Referenzen....153
11 Erklärung....156

1 Einleitung

In den letzten Jahren haben eine Reihe von Projekten unter dem Begriff „Web 2.0“ für Aufmerksamkeit gesorgt. Dazu zählen Web-Gemeinschaften, Weblogs aber auch eine Reihe von Webanwendungen und sogenannte Webservices. Webanwendungen sind Programme, die im „Web“, also in einem Webbrowser, ausgeführt werden. Eine Webanwendung muss weder auf einem Computer installiert, noch konfiguriert werden. Alles was man für eine Webanwendung braucht ist eine Internet-Verbindung und ein Webbrowser. Ein Beispiel für eine Webanwendung ist Google-Docs *[Google 1]*. Google-Docs ist eine Textverarbeitung und eine Tabellenkalkulation die im Webbrowser ausgeführt wird.

Im Bereich des E-Learnings spielt das Internet eine besondere Rolle. So werden zum Beispiel für den Fernstudiengang Informatik der Fachhochschule Trier eine Reihe von Kursen angeboten, die man über das Internet abrufen kann. Zum Studium gehört aber nicht nur das Lesen von Vorlesungsunterlagen oder Skripten sondern zum Beispiel auch Übungsaufgaben. So müssen im E-Learning-Kurs „Einführung in die Programmierung“ Java-Programme erstellt werden. An dieser Stelle stößt jedes E-Learning-System an seine Grenzen: um Java-Programme zu erstellen, bleibt den Lernenden nichts anderes übrig, als das E-Learning-System zu verlassen und andere Programme zu verwenden. Durch die modernen Techniken des „Web 2.0“ lässt sich dieses Problem beheben, indem mit einer Webanwendung E-Learning-Systeme um solch komplexe Aufgaben wie das Erstellen von Java-Programmen erweitert werden.

2 Ziele

Ziel dieser Abschlussarbeit ist das Erstellen einer Entwicklungsumgebung (IDE). Eine Entwicklungsumgebung ist eine Software, mit der Programme erstellt werden können. Aufgabe einer Entwicklungsumgebung ist es, Software-Entwickler beim Programmieren von Software unterstützen.

Die Entwicklungsumgebung, die im Rahmen dieser Abschlussarbeit erstellt werden soll, soll dabei für den Einsatz im E-Learning Bereich optimiert werden. Dies bedeutet insbesondere, dass die Entwicklungsumgebung einfach und intuitiv bedient werden kann. Eine weitere wichtige Voraussetzung für den Einsatz im E-Learning-Bereich ist die Online-Verfügbarkeit des Programms. Dies soll eine möglichst einfache Integration des Programms in bestehende, ebenfalls online verfügbare, E-Learning-Systeme ermöglichen.

Als Grundlage der zu entwickelnden Software (**Online-Compiler** oder kurz **OC**) soll dazu das Programm Java-Online-Compiler oder kurz JOC verwendet werden, welches im Rahmen einer Projektarbeit *[HV 2006]* entwickelt wurde. Der Java-Online-Compiler ist eine Entwicklungsumgebung für Java-Programme welche als Web-Service über jeden Web-Browser aufgerufen und ausgeführt werden kann. Im Gegensatz zum „normalen“ Entwicklungsumgebungen unterstützt der Java-Online-Compiler nur das Bearbeiten, Compilieren und Ausführen von Java-Programmen. Im Online-Compiler soll der JOC so erweitert werden, dass eine bessere Entwicklung von Programmen ermöglicht wird. Dazu muss, neben den Bearbeiten, Compilieren und Ausführen auch das Ausführen eines Programms mit einem Debugger zur Fehlersuche möglich sein. Außerdem sollen mehrere Programmiersprachen unterstützt werden.

Der Name Online-Compiler ist also eigentlich schlecht gewählt. Schließlich sollen nicht nur Dateien compiliert werden. Der Name stammt von dem „Java Online Compile Service“ *[SW 2005]*, einem Webservice, mit dem man Java-Dateien compilieren konnte. Diese Programm war Grundlage des Java-Online-Compilers welcher wiederum Grundlage des Online-Compilers sein soll.

3 Anforderungen und Konzept

Der Online-Compiler soll als reine Webanwendung implementiert werden. Das heißt, der Online-Compiler soll in jedem Webbrowser ausgeführt werden können, ohne das zusätzliche Software notwendig ist. Dies hat den Vorteil, dass Anwender weder Software installieren, noch konfigurieren müssen. Das Ausführen, Compilieren und Debuggen der Anwendungen soll auf einem Server erfolgen. Konfiguration oder Installation der verschiedenen Programmiersprachen sind für die Anwender also ebenfalls nicht nötig.

Wie der Java-Online-Compiler soll der Online-Compiler als Ajax-Webanwendung (Ajax = **A**synchronous **J**avaScript **a**nd **X**ML) implementiert werden. Eine „klassische“ - nicht-Ajax-Webanwendung - funktioniert nach folgendem Prinzip: Client-Anfragen werden von einem Webserver bearbeitet und das Ergebnis in Form einer HTML-Seite zum Client (Webbrowser) zurückgeschickt. Dieses Verfahren hat zwei Nachteile: zum einen ist die Entwicklung des Webanwendung aufwendig, da sehr viele HTML-Seiten erstellen werden müssen wobei sich leicht Fehler einschleichen können. Zum anderen ist das Verhalten des Servers aus Sicht des Anwenders statisch: Anfragen generieren HTML-Seiten, die dann komplett im Browser neu geladen werden (siehe Abbildung 1).

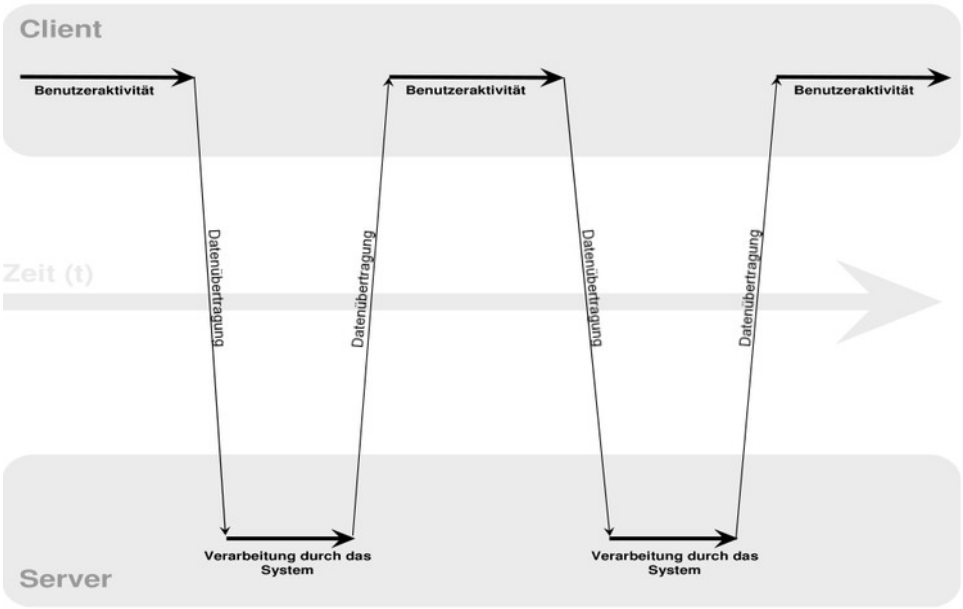


Abbildung 1: klassisches Modell einer Webanwendung: Benutzeraktivität erzeugt Anfragen an einen Server. Während der Server die Anfragen bearbeitet, muss der Client (Webbrowser) warten, bis er eine Antwort (HTML-Seite) vom Server bekommt.

Dieses Verhalten ist für Seiten mit statischen Inhalten zwar ausreichend, für eine komplexere Anwendung wie dem Online-Compiler, ist dieses „klassische“ Modell hingegen weniger geeignet.

Schließlich soll der Online-Compiler eine vollständige Entwicklungsumgebung für Programme sein. Also eine komplexere Webanwendung mit vielen, sich ändernden Inhalten und Funktionen, sodass viele Anfragen auf dem Server nötig sind. Mit dem klassischen Konzept würde dies bedeuten, dass bei jeder Anfrage, bei jeder Aktion des Anwenders, die gesamte Seite (das Webinterface des OC) neu geladen werden muss.

Im Unterschied zu „klassischen“ Webanwendungen werden bei Ajax-Webanwendung keine HTML-Dokumente bei jeder Client-Anfrage vom Webserver geladen. Bei Benutzeraktivität sendet die Webanwendung eine Anfrage zum Server der diese, meist mit einem XML-Dokument, beantwortet. Kommt eine Antwort bei einem Client an, werden die Inhalte der Webanwendung auf dem Client aktualisiert. Der große Vorteil ist dabei, dass während einer laufenden Client-Anfrage die Anwendung weiter vom Benutzer verwendet werden kann (siehe Abbildung 2). Man ist also nicht mehr darauf angewiesen, dass der Server ein HTML-Dokument erstellt und dieses beim Client ankommt. Zusammengefasst lässt sich sagen, Ajax-Webanwendungen verhalten sich mehr wie eine „normale“ Anwendung. Dadurch wird die Benutzerfreundlichkeit erhöht und lästige Wartezeiten fallen zum großen Teil weg.

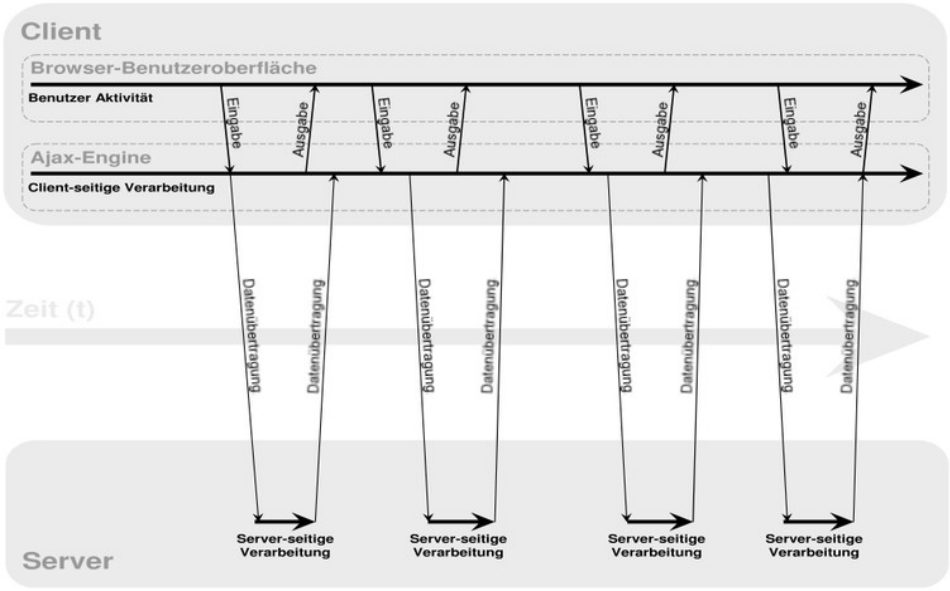


Abbildung 2: Modell einer Ajax-Webanwendung

Kennzeichnend für Ajax-Webanwendungen ist: der Server beantworten Anfragen nicht mehr mit HTML-Dokumenten sondern in Form von XML. Außerdem wird die Benutzeroberfläche (das Webinterface) wird mit dem Einsatz von JavaScript realisiert um ein dynamisches Verhalten zu ermöglichen.

Wie bei jeder Webanwendung wird für den Online-Compiler ein Server benötigt, der Anfragen von Clients (Web-Browsern) verarbeitet und beantwortet. Dazu soll, wie schon beim Java-Online-Compiler, ein so genanntes Java-Servlet verwendet werden. Ein Java-Servlet ist eine Java-Anwendung, mit der HTTP-Anfragen von Webbrowsern verarbeitet und beantwortet werden können. Ein Java-Servlet wird von einem sogenannten Servlet-Container, einem speziellem Web-Server ausgeführt. Für den Online-Compiler soll der Tomcat-Webserver als Servlet-Container verwendet werden. Die fertige Anwendung soll schließlich auf dem Linux-Server „javock“

(http://javock.fh-trier.de) installiert werden.

Der Online-Compiler soll zunächst nur für die beiden Webbrowser Firefox und Internet-Explorer entwickelt werden.

3.1 Allgemeine Funktionen

Nachdem im ersten Abschnitt die grundsätzlichen Anforderungen festgelegt wurden, wird in diesem Abschnitt näher auf einzelne Funktionen des Online-Compilers eingegangen.

- Unterstützung verschiedener Programmiersprachen:

  Mit dem Online-Compiler sollen Programme in verschiedenen Programmiersprachen entwickelt werden können. Die Programmiersprachen, die unterstützt werden sollen, sind Java, C und C++. Außerdem soll die Voraussetzung geschaffen werden, weitere Programmiersprachen wie C# oder Lisp zu einem späteren Zeitpunkt zu dem Online-Compiler hinzuzufügen.

- Erstellen und Verwalten von Dateien:

  Zum Entwickeln von Programmen gehört natürlich auch das Erstellen und Bearbeiten von Dateien und Verzeichnissen. Alle Dateien und Verzeichnisse, die von den Anwendern des Online-Compilers erstellt werden, sollen auf dem Server gespeichert werden. Auf diese Weise kann jeder Anwender von jedem Rechner mit einem Webbrowser direkt auf alle seine Dateien zugreifen. Um das Programmieren zu erleichtern, soll außerdem im Quelltext-Editor eine Syntax-Hervorhebung (Syntax-Highlight) erfolgen.

- weiter Anforderungen:

  Dokumentationen von Programmiersprachen sollen über den Online-Compiler geöffnet und durchsucht werden können. Der Online-Compiler soll verschiedene Sprachen unterstützen: deutsch, englisch und (teilweise) chinesisch.

3.2 Integration in Ilias

Die Hauptaufgabe des Online-Compilers soll der Einsatz im E-Learning Bereich sein. Deshalb müssen Möglichkeiten geschaffen werden, den Online-Compiler von dem E-Learning-System Ilias (http://ilias.fh-trier.de) aufzurufen. Insbesondere soll der Online-Compiler in die Online-Kurse „Einführung in die Programmierung“ und „Datenstrukturen + Algorithmen“ integriert werden. Dazu sind folgende Funktionen vorgesehen:

- Integration aller Programm-Beispiele (Java-Quelltext-Dateien) der Online-Kurse im Online-Compiler.
- Um das Verwalten und Bearbeiten von Programm-Beispielen so einfach wie möglich zu

gestalten, sollen die Programm-Beispiele, genauso wie alle anderen Dateien, über die Benutzerschnittstelle des Online-Compilers bearbeitet werden können. Dazu soll ein spezieller Administrator-Account verwendet werden.

- Aufruf der Programm-Beispiele im Online-Compiler über Links in den Online-Kursen
- Entwicklung einer speziellen Version der Benutzerschnittstelle des Online-Compilers für den Kurs „Datenstrukturen + Algorithmen“. Bei diesem Online-Kurse werden in einem kleinen Fenster Informationen (zum Beispiel Java-Applets) zu den geöffneten Kurs-Seiten dargestellt. Bei einigen Seiten soll der Online-Compiler mit einem geöffneten Programm-Beispiel in diesem Fenster dargestellt werden. Da der Platz dieses Fenster begrenzt ist, soll eine „kleinere“ Version der Benutzerschnittstelle des Online-Compilers („Ilias-Version“ oder „kleine Version“) erstellt werden
- Benutzer des Ilias-Servers sollen sich mit ihrem Ilias-Benutzernamen und dem dazugehörigen Passwort an den Online-Compiler anmelden können (Benutzerauthentifizierung über Ilias). Es soll allerdings auch noch die Möglichkeit bestehen, dass sich Nutzer als Gäste anmelden. Gäste benötigen kein Passwort und keinen Benutzernamen. Die Dateien, die Gäste erstellt haben, sollen nach deren Abmeldung wieder gelöscht werden.

3.3 Compiler und Linker

Quelltext-Dateien sollen mit dem Online-Compiler compiliert werden können. Dazu werden für die verschiedenen Programmiersprachen sogenannte **Backends** benötigt. Ein Backend ist ein Programm, welches im Hintergrund Aufgaben wie das Compilieren einer Datei vornimmt. Zum Compilieren von Java-Dateien soll, wie schon beim Java-Online-Compiler, der Eclipse-Java-Compiler (JDT) verwendet werden. Zum Compilieren von C- und C++-Dateien soll die GCC (GNU Compiler Collection) verwendet werden. Für eine spätere Implementierung einer C#-Unterstützung soll Mono *[Novell 2]* verwendet werden.

C- und C++-Programme müssen nicht nur compiliert sondern auch noch mit einem Linker verarbeitet werden. Dazu müssen die Dateien, die mit einem Linker zu einem Programm „zusammengebunden“ werden sollen, angegeben werden. Der Online-Compiler muss es dem Benutzern also erlauben, für C- und C++-Dateien festzulegen, welche Dateien (sogenannte Linker-Targets oder einfach nur **Targets**) mit einem Linker zu einem Programm hinzugefügt werden sollen.

3.4 Debugger

Der Online-Compiler soll das Debuggen von Programmen ermöglichen. Dazu müssen einige Funktionen vom Online-Compiler unterstützt werden:

- Debugger-Haltepunkte (Breakpoints) müssen zu Dateien hinzugefügt werden können. Wenn ein Programm eine Zeile mit einem Haltepunkt erreicht, soll das Programm in der Zeile anhalten.
- Die Zeile und die Datei, in der sich das Programm gerade befindet, sollen dem Benutzer angezeigt werden.

– Der Anwender soll die Möglichkeit haben, Werte von Variablen eines Programmes, welches durch den Debugger ausgeführt wird, anzuzeigen (Debugger-Variablen).

3.5 Sicherheitskonzept

Mit dem Online-Compiler sollen Anwender die Möglichkeit haben, Programme zu entwickeln und auf dem Server auszuführen. Damit die Sicherheit der Servers durch die Anwender-Programme nicht gefährdet wird, müssen einige Sicherheitsvorkehrungen getroffen werden. So dürfen Anwender-Programme zum Beispiel nicht Dateien löschen oder den Rechner herunterfahren. Dass heißt, einige Funktionen müssen für Anwender-Programme blockiert werden.
Dazu soll für Java-Programme der sogenannte Java-Security-Manager verwendet werden. Der Java-Security-Manager ist eine Erweiterung der Java-VM (Virtuall Machine), mit dem man bestimmte Befehle wie das Löschen von Dateien für Java-Programme sperren kann.
Für C- und C++-Programme soll die Anwendung AppArmor *[Novell 1]* verwendet werden. Diese erlaubt es, ähnlich dem Java-Security-Manager, bestimmte Befehle für eine Anwendung zu sperren.

3.6 MJVM

Während der Entwicklung des Java-Online-Compilers wurde festgestellt, dass das Ausführen von Java-Programmen relativ langsam ist. Durch mehrere Tests wurde weiterhin festgestellt, dass die meiste Zeit beim Starten der Java-VMs (Virtuall Machine), die die Java-Programme ausführen sollen, verloren geht. Der beste Weg, die Ausführung von Java-Programmen zu beschleunigen, ist deshalb die Zeit, die zum Starten von Java-VMs benötigt wird, zu verringern. Dies kann zum Beispiel dadurch erreicht werden, in dem mehrere Java-Anwendung in einer einzigen Java-VM ausgeführt werden. Die Zeit zum Starten der Java-VMs fällt dadurch praktisch weg. Im Rahmen dieser Abschlussarbeit soll geprüft werden, ob diese Lösung (MJVM = Multi-Task-Java-VM) praktisch realisierbar ist und ob damit der erhoffte Geschwindigkeitsvorteil erreicht werden kann.

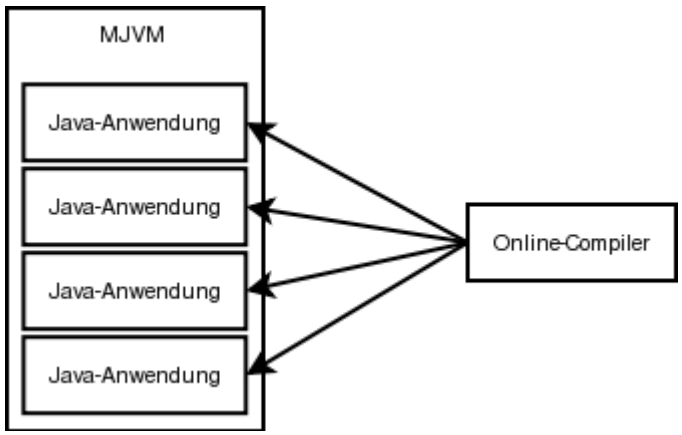


Abbildung 3: Konzept MJVM

3.7 Zusammenfassung

Ziele dieser Abschlussarbeit im Überblick:

- **Entwurf, Implementierung und Test einer Webbasierten Entwicklungsumgebung:**
  - Web-Anwendung (Ajax) mit einem Webinterface und einem Servlet (Tomcat-Server)
  - Unterstützung verschiedener Sprachen für die Benutzerschnittstelle
  - Unterstützung verschiedener Programmiersprachen (Java, C, C++)
  - Erweiterungsmöglichkeit für weitere Programmiersprachen (insbesondere Lisp und C#)
  - Compilieren von Java-Dateien (Eclipse Java-Compiler)
  - Compilieren und Linken von C- und C++-Dateien (GNU Compiler Collection)
  - Ausführen von Java-Programmen (SUN Java-VM)
  - Ausführen von C- und C++-Programmen (mit AppArmor)
  - Debuggen von Java-Programmen
  - Debuggen von C- und C++-Programmen (GNU Debugger)
  - Debugger: Haltepunkte, Variablen der Programme die debuggt werden anzeigen (Debugger-Variablen)
  - Dateiverwaltung: erstellen, löschen, speichern, umbenennen usw. von Dateien und Verzeichnissen
  - Benutzerverwaltung
  - Authentifizierung über Ilias-Server
  - Gast-Benutzer (Anmeldung ohne Name und Passwort)
  - Administrator-Benutzer die Beispiele und ggf. auch Übungsaufgaben erstellen und verwalten können
  - Syntax-Hervorhebung im Quelltexteditor
  - Programmiersprachen-Dokumentationen durchsuchen
  - Integration in Ilias (Aufruf von Programm-Beispielen im Online-Compiler)
  - „kleine“ Ilias-Version des Online-Compilers

- **Entwurf, Implementierung und Test der Java-Anwendung MJVM zum Beschleunigten Ausführen von Java-Anwendungen:**
  - Java-Anwendungen sollen parallel in einer einzigen Java-VM (der MJVM) ausgeführt werden können

4 Entwurf

4.1 Online-Compiler

Der Online-Compiler ist eine klassische Client-Server-Anwendung:

- ein **Client**, auf dem der Benutzer Programme erstellen kann (Webinterface bzw. Benutzerschnittstelle)
- ein **Server**, auf dem die Programme der Benutzer compiliert und ausgeführt werden (Java Servlet)

Das Design des Online-Compilers ist deshalb mit dem des Java-Online-Compilers identisch:

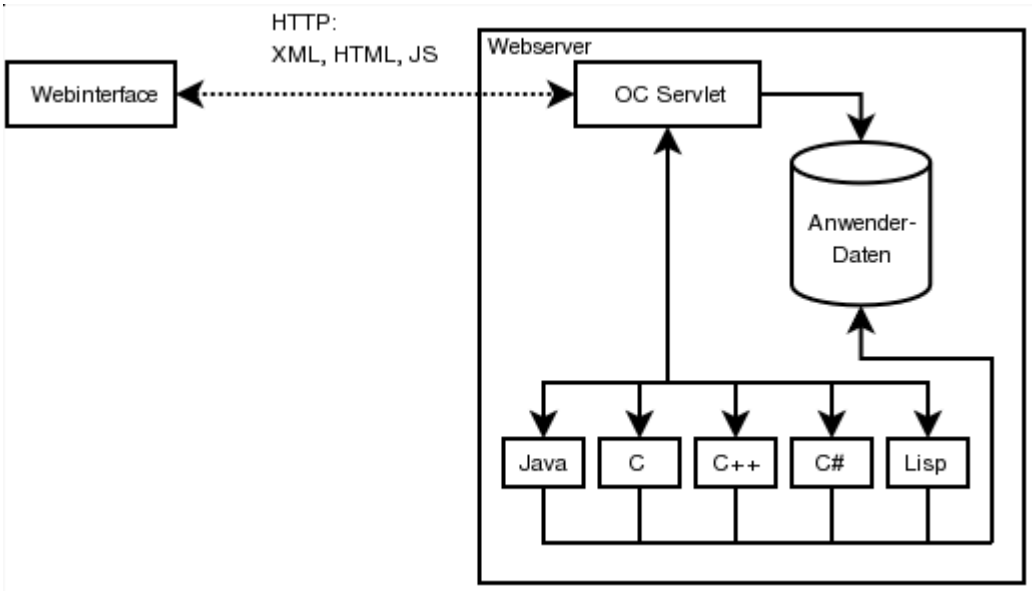


Abbildung 4: Aufbau Online-Compiler

- der Server beantwortet Anfragen, die er von den Clients erhält.
- Programme werden auf dem Client bearbeitet und, um sie zu compilieren, auszuführen und Fehler zu suchen (Debugger), auf dem Server gespeichert.
- wie schon im Java-Online-Compiler soll auch der Online-Compiler als AJAX-Anwendung entwickelt werden. Das heißt, im Gegensatz zu anderen Web-Anwendungen werden vom Server (bzw. vom Servlet) keine HTML-Seiten generiert und einfach auf dem Client dargestellt, sondern XML-Antworten zu einem AJAX-Client gesendet, der diese auf der Webseite (der Benutzerschnittstelle) anzeigt.

4.1.1 Servlet

Wie bereits erwähnt, besteht der Online-Compiler aus zwei Anwendungen: einer Benutzerschnittstelle und dem Servlet. Während die Benutzerschnittstelle die Aufgabe hat, dem Anwender den Zugriff auf die Funktionen des Online-Compilers zu ermöglichen, ist es die Aufgabe des Servlets, diese Funktionen auszuführen. Die einzelnen Anforderungen, die an das Servlet gestellt werden, wurden bereits am Anfang vorgestellt.

Aufgaben des Servlets:

- Verwalten der Quelltextdateien, die von Anwendern erstellt wurden
- Verwalten von Beispielen und gegebenenfalls später auch die Verwaltung von Übungsaufgaben
- Verwalten von Benutzern, unter anderem mit Hilfe von anderen Systemen (Ilias-Server *[Ilias]*)
- Bereitstellen einer Schnittstelle für Programmiersprachen, um Programme zu erstellen (Compilieren und Linken), auszuführen und, wenn möglich, zu debuggen bzw. sie mit einem Debugger auszuführen
- Bereitstellen einer Schnittstelle, die es Clients erlaubt, Anfragen zu stellen

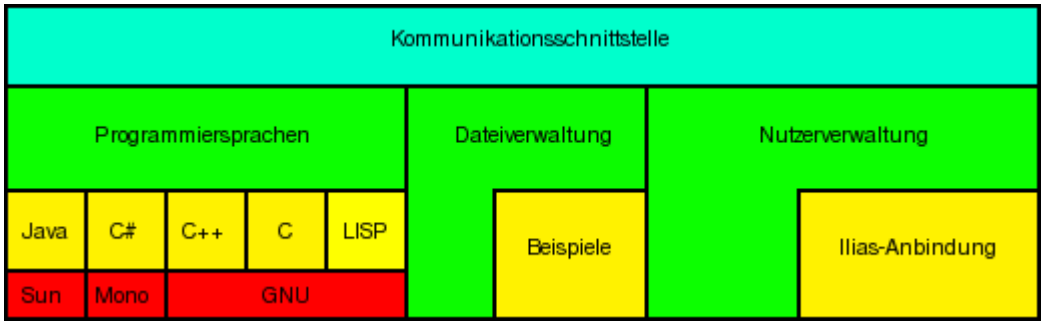


Abbildung 5: Module des Online-Compiler-Servlets

Auf die einzelnen Bestandteile des Servlets wird in den folgenden Abschnitten detailliert eingegangen.

4.1.1.1 Dateiverwaltung

Eine wichtige Aufgabe des Servlets ist die Speicherung und Verwaltung aller Quelltextdateien, die von den Anwendern erstellt werden. Das folgende UML-Diagramm zeigt die Klassenhierarchie aller Klassen, die zum Speichern von Dateien und Verzeichnissen dienen.

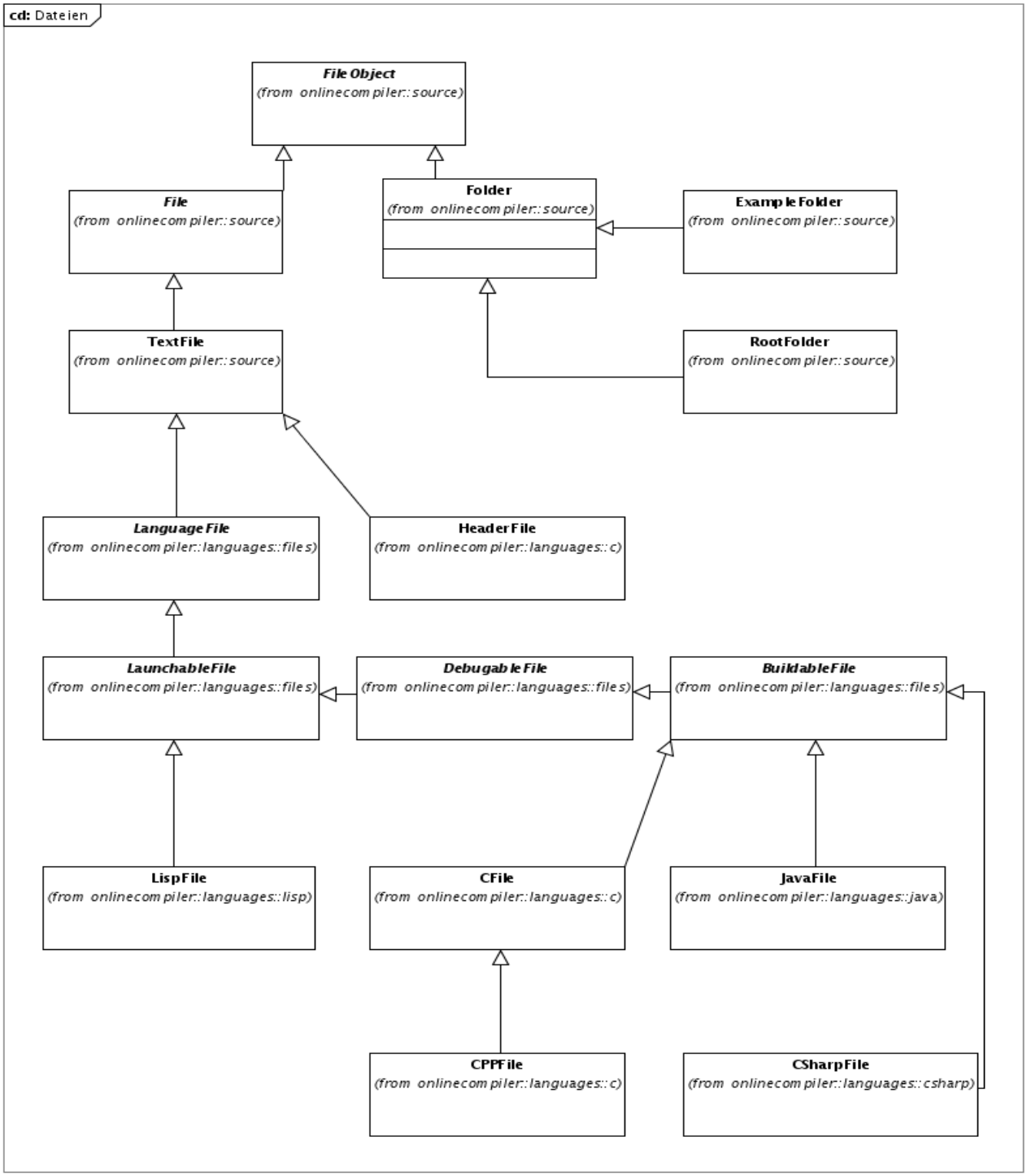


Abbildung 6: Klassenhierarchie Dateien und Verzeichnisse

Jedes Objekt einer Klasse stellt dabei eine Datei oder ein Verzeichnis dar. Die Klassen selbst sind hingegen die Typen der Dateien bzw. Verzeichnisse.

Die Klasse *FileObject* ist Super-Klasse aller Datei- und Verzeichnis-Klassen. In ihr sind Attribute und Methoden enthalten, die für alle anderen Datei- und Verzeichnis-Klassen wichtig sind. Dazu zählen zum Beispiel der Name der Datei bzw. des Verzeichnisses oder das Verzeichnis, in dem sich die Datei bzw. das Verzeichnis befindet.

Von den Datei-Klassen sind besonders die drei Klassen *LaunchableFile*, *DebugableFile* und *BuildableFile* wichtig. Diese drei Klassen bilden die Basis für nahezu alle Quelltext-Datei-Typen. *LaunchableFile* ist Basis-Klasse für alle Dateien, die ausgeführt werden können bzw. die ein Ausführbares Programm enthalten. *DebugableFile* ist Basis-Klasse für alle Dateien, die mit einem Debugger aufgerufen werden können. Dass heißt, sie müssen ebenfalls ein ausführbares Programm enthalten. *BuildableFile* ist schließlich Basis-Klasse für alle Dateien, die compiliert und oder gelinkt werden können. Von diesen drei Super-Klassen sind fast alle Quelltext-Dateien abgeleitet: *LispFile* für Lisp-Programme die weder compiliert noch debuggt werden können, *JavaFile* für Java-Klassen bzw. Java-Programme, *CSharpFile* für C#-Klassen bzw. C#-Programme, *CFile* für C-Quelltexte und *CPPFile* für C++-Quelltexte. Ausnahme ist nur die Klasse *HeaderFile* in der C- und C++-Headerdateien gespeichert werden. Diese Dateien können nicht ausgeführt, compiliert oder debuggt werden sondern enthalten nur Text.

Klassen für Verzeichnisse sind *Folder*, *RootFolder* und *ExampleFolder*. Instanzen der Klasse Folder sind normale Verzeichnisse während *RootFolder* und *ExampleFolder* spezielle Verzeichnisse darstellen. Die Klasse *RootFolder* dient als Wurzel-Verzeichnis, in dem sich alle anderen Verzeichnisse und Dateien eines Anwenders befinden. Im Gegensatz zu allen anderen Dateien und Verzeichnissen sollen *RootFolder*-Instanzen kein übergeordnetes Verzeichnis haben. Die Klasse *ExampleFolder* dient zum Speichern von Programm-Beispielen.

4.1.1.2 Programmiersprachen

Eine wichtige Anforderung an den Online-Compiler ist die Unterstützung mehrerer Programmiersprachen. Die vorherige Version JOC (Java-Online-Compiler) erlaubt nur das Entwickeln von Java-Programmen. Der Online-Compiler soll hingegen neben Java auch C, C++, C# und Lisp unterstützen. Zudem soll eine spätere Integration anderer Programmiersprachen (zum Beispiel Prolog) so einfach wie möglich sein. Dies kann durch den Einsatz objektorientierter Konzepte wie Vererbung realisiert werden.

Der entscheidende Unterschied zwischen allen Sprachen sind die Backends, also die Programme, die das eigentliche Compilieren, Ausführen und Debuggen von Programmen durchführen. So wird für Java-Programme die Java-VM von SUN *[Sun 1]* verwendet während C und C++ die Programme der GNU-Compiler-Collection (GCC) verwenden *[FSF 1]*.

Das folgende UML-Diagramm zeigt die Hierarchie der Klassen an, die die Unterstützung der einzelnen Programmiersprachen liefern.

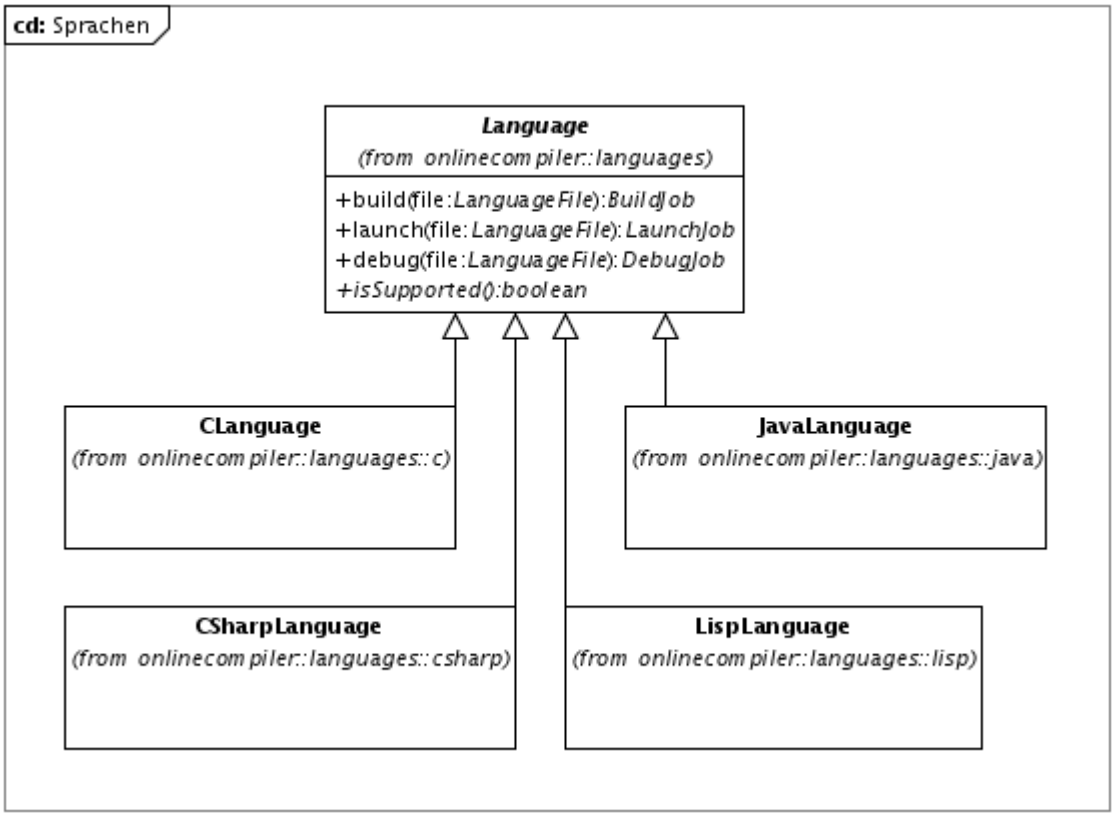


Abbildung 7: Klassenhierarchie Programmiersprachen

Die wichtigste Klasse ist *Language*. Diese abstrakte Klasse ist Basis aller anderen Klassen. Von ihr sind 4 Klassen für die 5 Programmiersprachen C, C++, C#, Java und Lisp abgeleitet. Da sich C und C++ kaum unterscheiden (zumindest was das Compilieren und Linken betrifft) reicht für beide Sprachen eine Klasse aus. Die Methoden der Klassen *Language* erlauben es, Programme zu erzeugen (Methode *build*), Programme auszuführen (Methode *launch*) und Programme mit einem

Debugger auszuführen (Methode *debug*). Die Implementierung dieser Methoden erfolgt in den abgeleiteten Klassen für die jeweilige Programmiersprache und das jeweilige Backend, welches dazu verwendet wird. So muss zum Beispiel in der Methode *launch* der Klasse *JavaLanguage* die JVM aufgerufen werden, während in der Methode *launch* der Klasse *LispLanguage* der Lisp-Interpreter aufgerufen werden muss.

4.1.1.3 Jobs: Compilieren, Ausführen und Debuggen

Das Ausführen von Programmen, das Compilieren von Quelltext-Dateien oder das Debuggen von Programmen erfordert einigen Aufwand und vor allem Zeit. Dies bedeutet zum Beispiel, dass ein Anwender eine gewisse Zeit warten muss, bis das Compilieren einer Datei abgeschlossen ist. Normalerweise würde der Server beziehungsweise das Servlet dann solange blockiert sein (bzw. keine weiteren Anfragen mehr beantworten), bis das Compilieren abgeschlossen ist. Da solch ein Verhalten nicht akzeptabel ist, kann man schon beim Entwurf festlegen, dass alle Aufgabe die längere Zeit in Anspruch nehmen, in einem eigenen Thread ausgeführt werden. Diese Threads werden als **Job** bezeichnet.

In dem folgenden URL-Diagramm wird die Klassenhierarchie für solche Threads (Jobs) dargestellt:

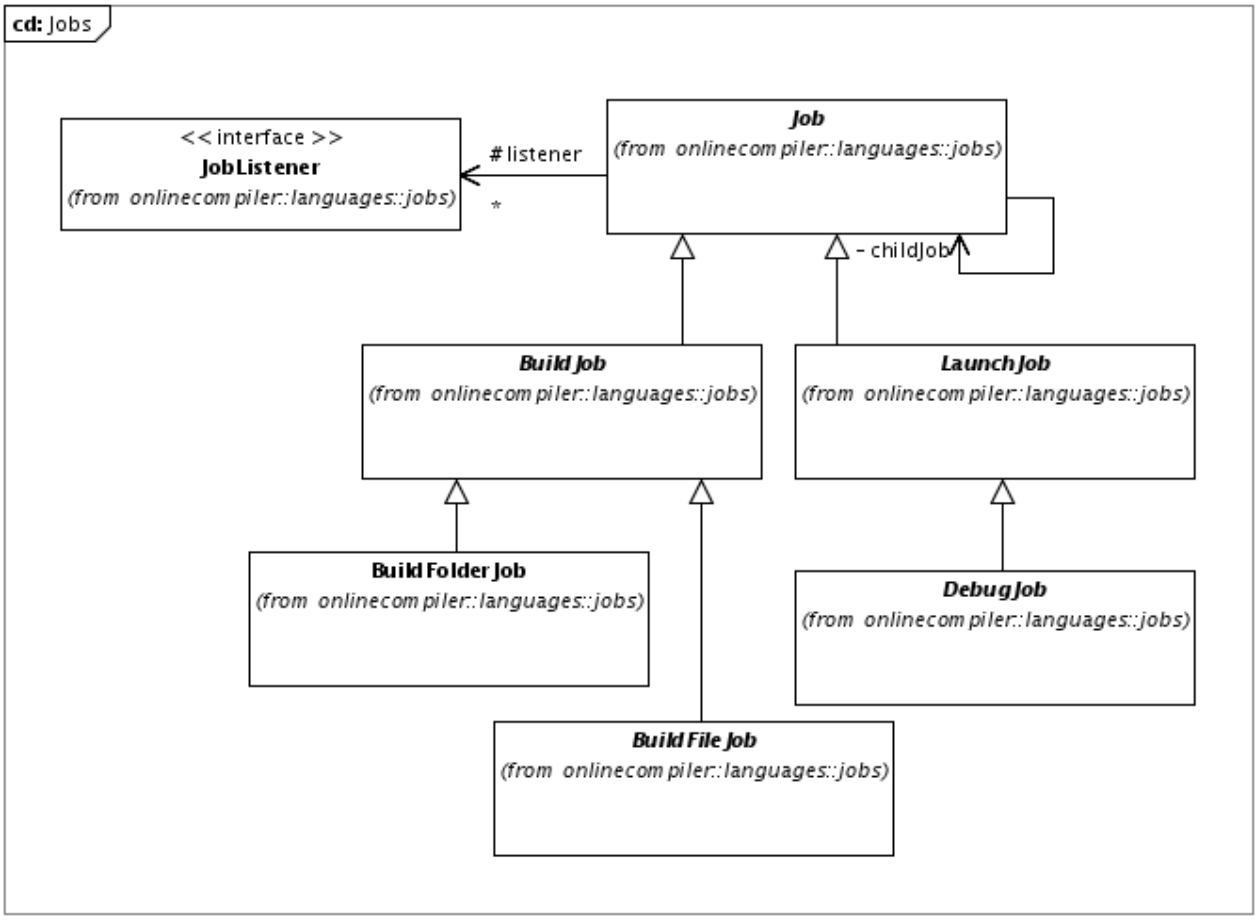


Abbildung 8: Klassenhierarchie Jobs

Alle Jobs sind von der Klasse *Job* abgeleitet. Die Klasse *Job* hat eine Assoziation zur Schnittstelle *JobListener*. Auch hier wurde, wie im gesamten Projekt, das Observer-Entwurfsmuster verwendet. Die *Job*-Klasse übernimmt dabei die Funktion eines Modells. Observer-Objekte (Listener), die von der Klasse *JobListener* abgeleitet sind, können sich am Modell bzw. am Job anmelden. Bei bestimmten Ereignissen werden die angemeldeten Listener vom Modell bzw. dem Job benachrichtigt. Ereignisse sind zum Beispiel: Starten eines Jobs, Fehlermeldungen, Abbruch eines Jobs oder Beenden eines Jobs. Listener sind zum Beispiel die Benutzerschnittstellen (Webinterface).

Eine weitere wichtige Eigenschaft der Klasse *Job* ist die Assoziation *childJob*. *childJob* - also Kind-Job – stellt einen weiteren Job dar, der innerhalb eines anderen Jobs ausgeführt wird. Dies bedeutet, dass ein Job aus mehreren anderen Jobs bestehen kann. Zum Beispiel kann man so vor dem Ausführen einer Datei (*LaunchJob*) die Datei mit einem *BuildJob* Compilieren. Der BuildJob wäre dann der *childJob* wobei dieser natürlich weitere Kind-Jobs enthalten kann.

Von der Super-Klasse *Job* sind verschiedene Klassen abgeleitet, die die verschiedenen Job-Typen repräsentieren:

- *BuildJob*: Compilieren und/oder Linken eines Programmes. Die davon abgeleitete Klasse *BuildFolderJob* dient zum Compilieren und ggf. zum Linken aller Dateien in einem bestimmten Verzeichnis.
- *LaunchJob*: Ausführen eines Programmes
- *DebugJob*: Debuggen eines Programmes

Wie bereits im vorherigen Kapitel beschrieben, ist die Implementierung einzelner Funktionen abhängig von der jeweiligen Programmiersprache. Die „Job-Klassen“ müssen deshalb mehrmals, für jede Programmiersprache, implementiert werden. Es muss also von jeder Job-Klasse für jede Programmiersprache eine Klasse abgeleitet werden.

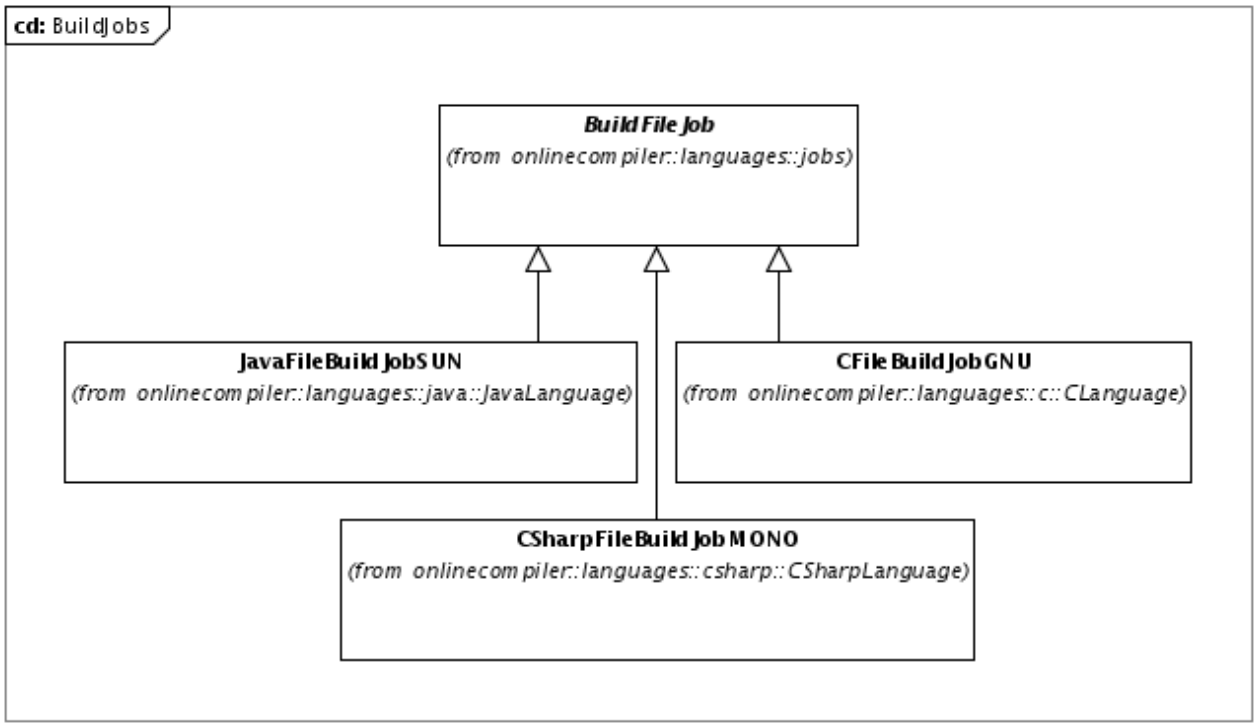


Abbildung 9: Klassen zum Compilieren und Linken von Programmen

Für die Namen der Klassen gilt folgende Konvention:

- Der vordere Teil des Namens beschreibt den Dateiinhalt, der mit dem Job compiliert, debuggt oder ausgeführt werden kann (z.B. JavaFile für Java oder CFile für C).
- Der mittlere Teil des Namens beschreibt die Art des Jobs (z.B. *BuildJob* oder *LaunchJob*).
- Der hintere Teil des Namens bezeichnet das verwendete Backend (z.B. GNU oder SUN). Wenn zum Beispiel mehrere verschiedene Backends für Java verwendet werden sollen (z.B. GNU und SUN), könnte man dazu verschiedene Klassen erstellen, deren Name jeweils mit GNU bzw. SUN endet.

Wenn man das obere UML-Diagramm betrachtet, fällt auf, dass nur Klassen für Java, C (und C++) und C# existieren. Für die Programmiersprache Lisp wird keine Implementierung der Klasse *BuildJob* benötigt, da Lisp eine Interpreter-Sprache ist.

Im nächsten UML-Diagramm werden die Klassen dargestellt, welche die Klasse *LaunchJob* implementieren. Diese Klassen bzw. Jobs dienen zum Ausführen von Programmen.

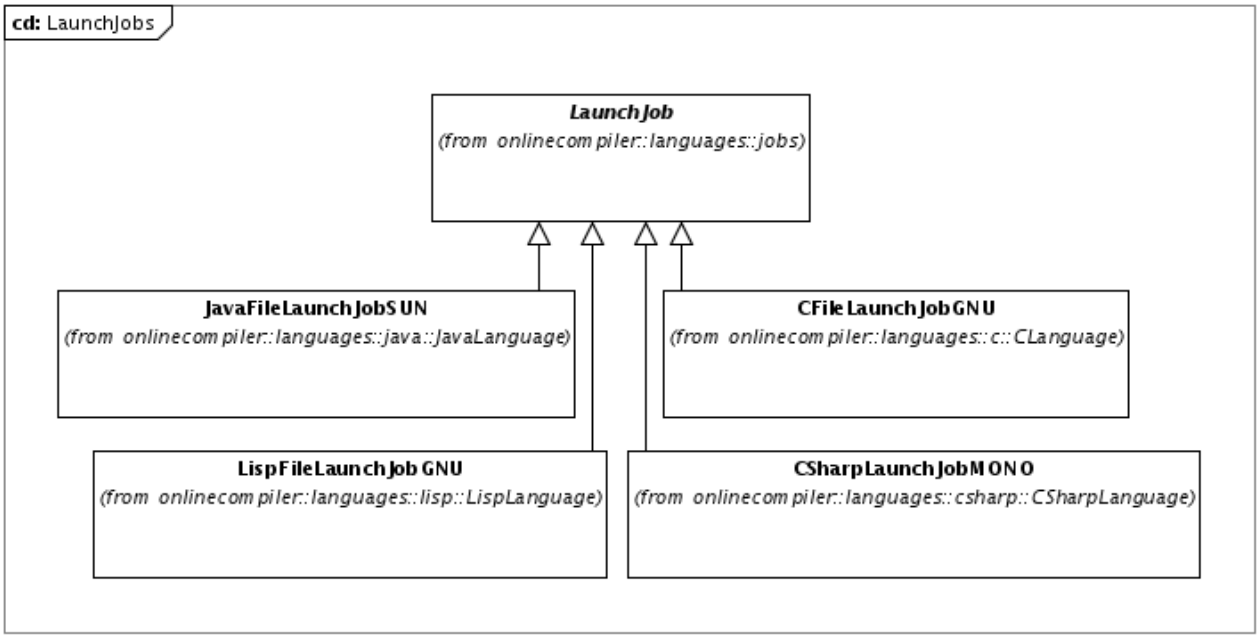


Abbildung 10: Klassen zum Ausführen von Programmen

Das folgende UML-Diagramm zeigt die Klassen, welche die Klasse *DebugJob* implementieren. Wie bei den *BuildJob*-Klassen gibt es auch bei den *DebugJob*-Klassen keine Implementierung für die Programmiersprache Lisp.

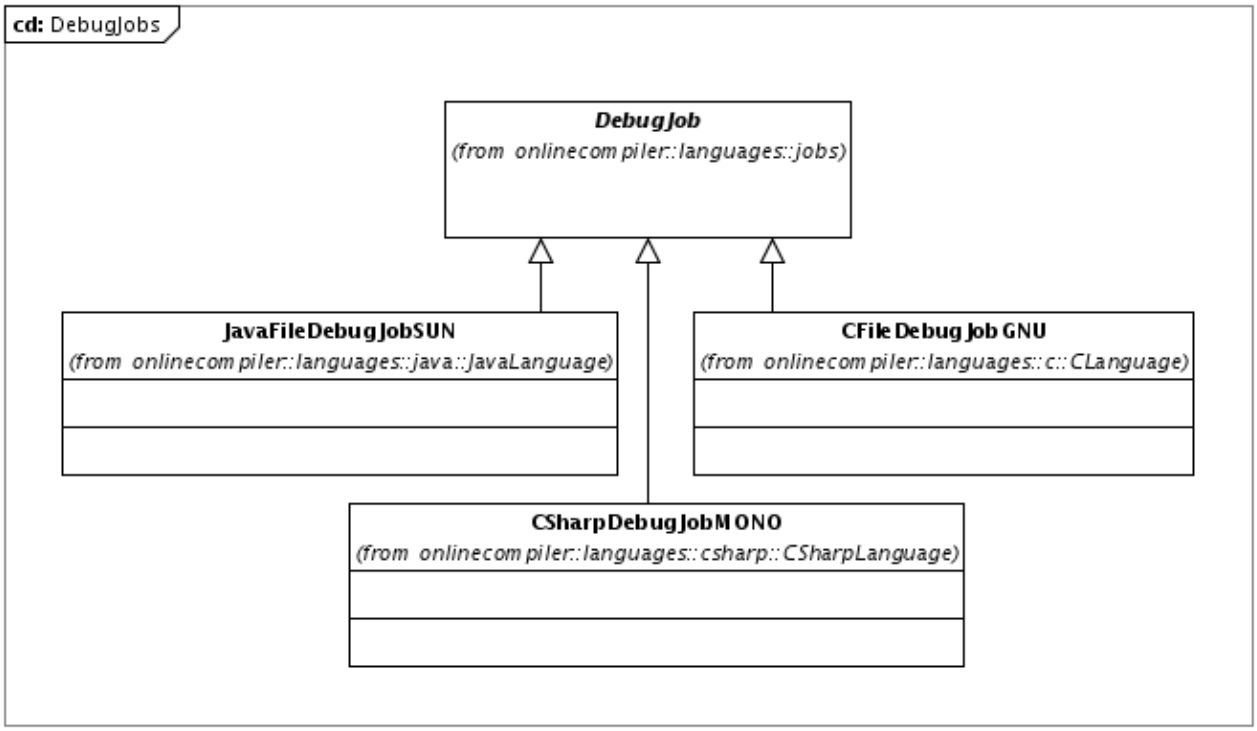


Abbildung 11: Klassen zum Debuggen von Programmen

4.1.1.3.1 Linker

Der Online-Compiler soll verschiedene Programmiersprachen unterstützen. Diese unterscheiden sich nicht nur in dem bereits mehrfach erwähnten Backend, welches zum Compilieren und Ausführen verwendet wird, sondern auch in der Art, wie Programme erzeugt werden müssen. Drei verschiedene Möglichkeiten können vorkommen:

- Dateien müssen nicht compiliert werden (z.B. LISP)
- Dateien müssen nur compiliert werden (z.B. Java und C#)
- Dateien müssen compiliert werden, mehrere compilierte Dateien werden mit einem sogenannten Linker zum einem Programm „zusammengebunden“ (C und C++). In solchen Fällen muss gespeichert werden, welche Dateien „gelinkt“ werden sollen.

In dem folgenden UML-Diagramm wird eine N:N-Beziehung der Klasse *BuildableFile* mit sich selbst dargestellt. Jede Datei kann zu einer oder mehren Dateien „gelinkt“ werden. Jede Datei kann aber auch aus mehren anderen Dateien bestehen, die zu ihr „gelinkt“ werden müssen. Wie bereits im Kapitel „Anforderungen und Konzept“ erwähnt wurde, werden Dateien, die zu anderen Dateien gelinkt werden, als Targets (Ziele) bezeichnet.

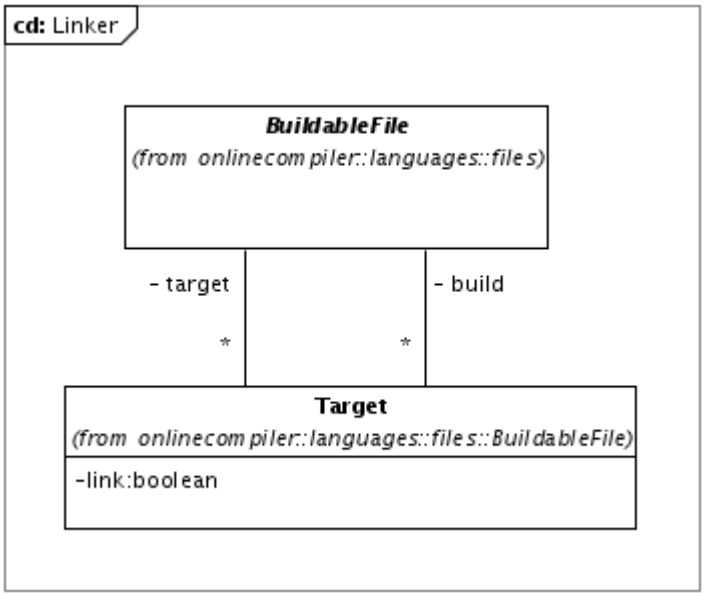


Abbildung 12: Targets und Dateien

Um überall das selbe Konzept zu verwenden, werden Targets auch bei Programmiersprachen verwendet, die keinen Linker verwenden. Dass heißt auch zu Java- und C#-Dateien können Targets angegeben werden. Diese werden dann nicht zu einem Programm „gelinkt“ sondern lediglich compiliert.

4.1.1.4 Nutzerverwaltung

In den ersten Abschnitten wurden Dateien und Verzeichnisse und deren Verarbeitung durch den Online-Compiler behandelt. Neben der Dateiverwaltung ist eine weitere wichtige Funktion des Online-Compilers die Verwaltung von Benutzern. Unter Verwaltung von Benutzern versteht man das An- und Abmelden von Benutzern (ggf. durch Überprüfung von Passwort und Name) sowie das Speichern der Quelltextdateien der einzelnen Nutzer. Wie bereits im Kapitel „Anforderungen und Konzept“ beschrieben, gibt es drei verschiedene Arten von Benutzern:

- Benutzer, die über einen Ilias-Server authentifiziert werden.
- Benutzer, die administrative Aufgaben haben. Dazu zählt zum Beispiel das Verwalten der Programm-Beispiele und ggf. der Übungsaufgaben
- Gast-Benutzer, die sich ohne Passwort anmelden. Bei solchen Benutzern werden keine Quelltextdateien gespeichert. Dass heißt, nach dem Abmelden werden alle Daten des Benutzer gelöscht.

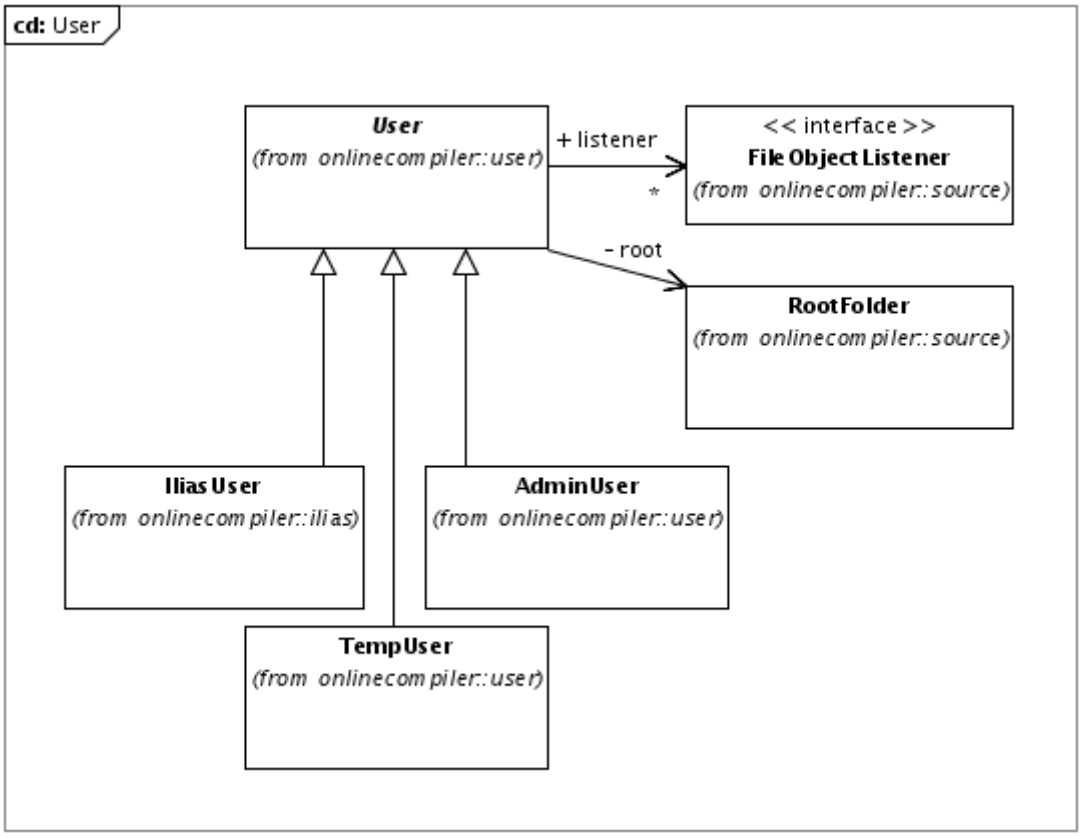


Abbildung 13: Klassenhierarchie Online-Compiler-User

Basis-Klasse aller Benutzer-Klassen ist *User*. Die abgeleiteten Klassen *IliasUser*, *AdminUser* und *TempUser* stellen die drei verschiedenen Benutzer-Typen dar. Die Unterschiede zwischen den einzelnen Benutzer-Typen wurden bereits im Kapitel „Anforderungen und Konzept“ vorgestellt:

- IliasUser: Passwort und Benutzername werden mit einem Ilias-Server überprüft (Authentifizierung über einen Ilias-Server)
- AdminUser: Passwort und Benutzername des Administrators
- TempUser: Anmeldung ohne Passwort und Benutzername (Gäste), wurde bereits im Java-Online-Compiler verwendet

Im obigen UML-Diagramm ist die Schnittstelle *FileObjectListener* enthalten. Diese Schnittstelle ist Basis-Klasse für Listener-Objekte. An User-Objekten kann man also Listener-Objekte (Entwurfsmuster Observer) anmelden. Die Listener-Objekte werden bei Ereignissen, die Dateien des Nutzers betreffen, informiert. Das User-Objekt ist also nicht das eigentliche Modell sondern dessen Dateien und Verzeichnisse. Die Dateien und Verzeichnisse können über die Beziehung *root* erreicht werden (siehe Kapitel „Dateiverwaltung“).

4.1.1.5 Servlet-Infrastruktur

Der letzte Teil des Entwurfs des Online-Compiler-Servlets beschäftigt sich mit dem Teil, der die Servlet-Funktionalität bereitstellt. Dessen Entwurf ist vorgegeben. So muss jedes Servlet eine zentrale Klasse haben, die zum Beantworten von Client-Anfragen dient. Die Klasse *OCServlet* übernimmt diese Aufgabe. Eine weitere vorgegebene Klasse ist *GarbageCollector*. Diese Klasse dient zum Löschen von Client-Verbindungen, die längere Zeit nicht verwendet wurden.

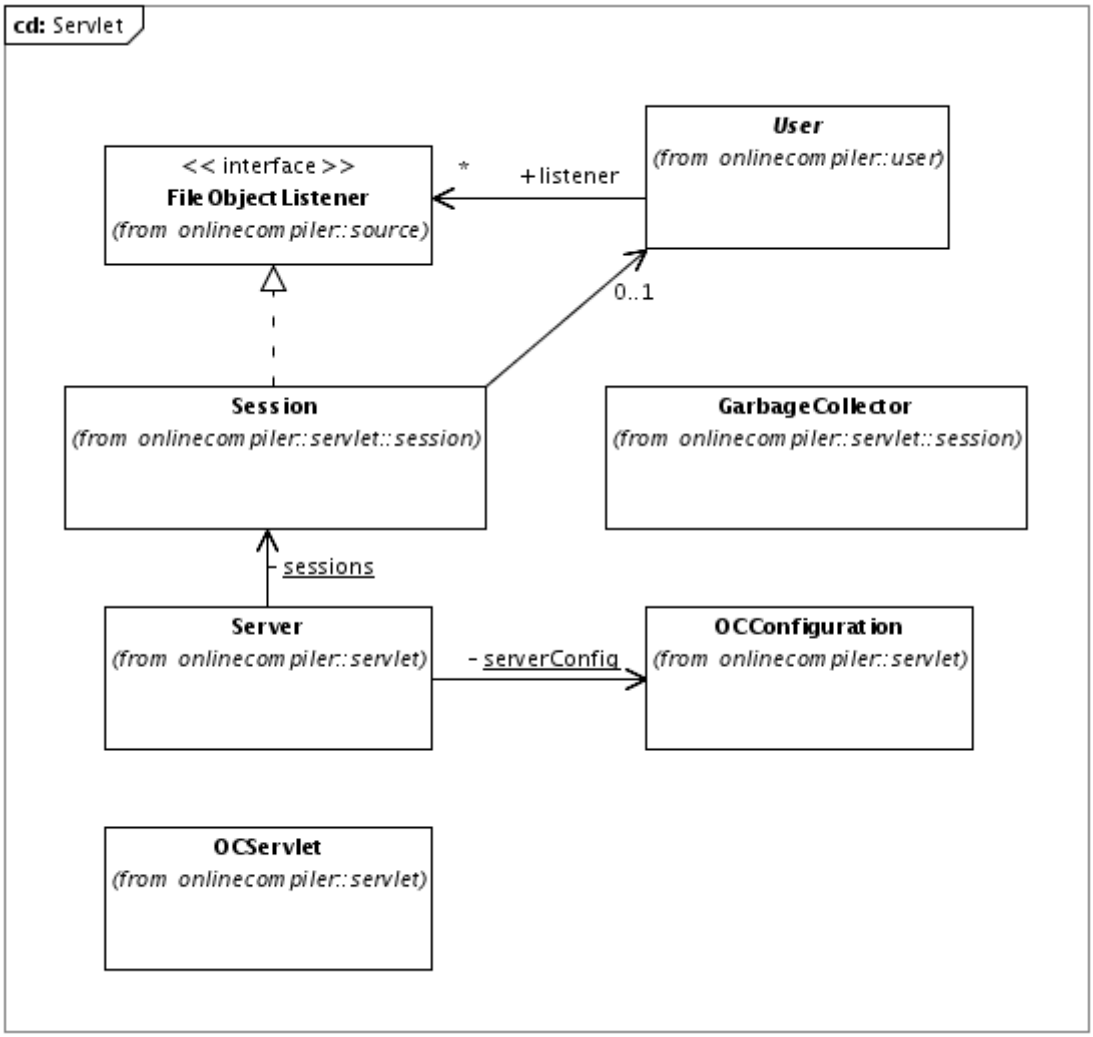


Abbildung 14: Klassenhierarchie Servlet und Sessions

Zentrale Klasse des Servlets ist *Server*. Diese Klasse dient als Container aller Client-Verbindungen (Klasse *Session*), zum Speichern der Konfiguration (Klasse *OCConfiguration*) sowie zum Initialisieren aller Anwendungsteile. Die Klasse *Session* dient zum Verwalten aller Client-Verbindungen. Dass heißt, zu jeder Verbindung mit einem Client existiert eine Instanz der Klasse *Session* um den Zustand der Verbindung zu speichern. Dazu gehört auch ein User-Objekt, sofern der Anwender des Client sich angemeldet hat. Die Klasse Session implementiert die Schnittstelle *FileObjectListener* um Ereignisse des Users (zum Beispiel Änderungen von Dateien) zu erfassen.

Diese Ereignisse müssen an den Client übertragen werden, um dort angezeigt zu werden.

Die letzte Anforderung, die in der Entwurfsphase des Servlets berücksichtigt werden muss, ist die der Mehrsprachenunterstützung für die Benutzeroberfläche. Es gibt grundsätzlich 3 Möglichkeiten, dies zu realisieren:

1. Für jede Sprache eine eigene Benutzeroberfläche erstellen. Das heißt zum Beispiel, das für jede Sprache eigene HTML-Dateien erstellt werden. Diese Methode hat den Nachteil, dass bei Änderungen die Dateien aller Sprachen geändert werden müssen.
2. Textinhalte der Benutzeroberfläche dynamisch zur Laufzeit auf der Client-Seite anpassen. Der Client ersetzt Texte in der Benutzeroberfläche durch entsprechende Texte der aktuellen Sprache. Dieses Verfahren hat den Nachteil, dass die Ausführungsgeschwindigkeit der Benutzeroberfläche verlangsamt wird, da im Hintergrund Texte ersetzt werden müssen. Außerdem muss das Übersetzen bei jedem Neuladen der Benutzeroberfläche wiederholt werden.
3. Die dritte und beste Methode ist das dynamische Ändern der Dateien zur Laufzeit auf Server-Seite, dass heißt durch das Servlet. Diese Methode soll beim Online-Compiler verwendet werden.

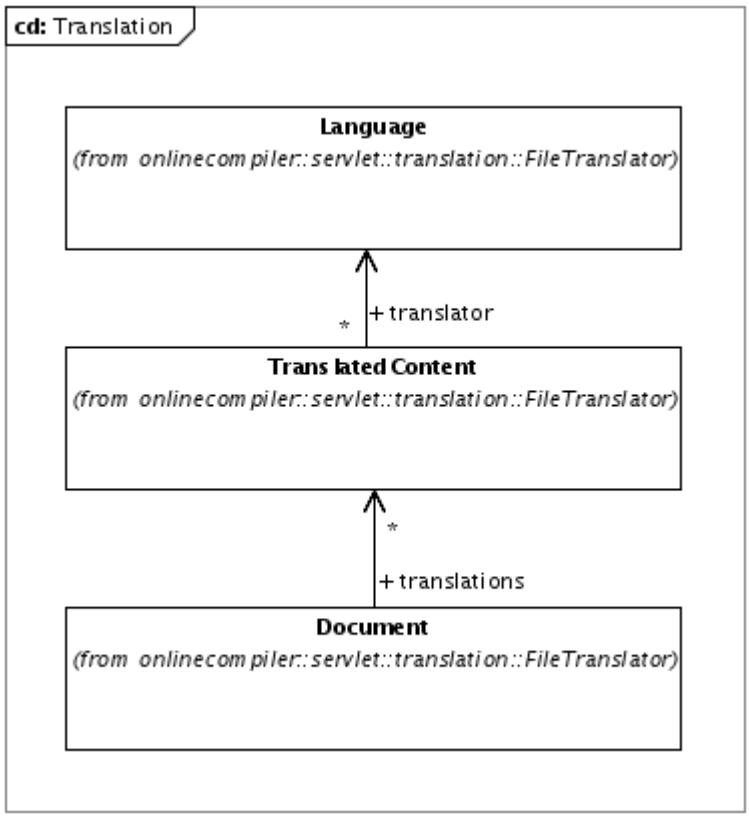


Abbildung 15: Klassen zum Übersetzen der Benutzerschnittstelle

Das obige UML-Diagramm zeigt die Klassen, die zum Übersetzen der Benutzeroberfläche in verschiedene Sprache benötigt werden. Die Klasse *Language* enthält die eigentliche Übersetzung für eine Sprache. Dazu wird eine Liste von Texten und den dazugehörigen Übersetzungen in einem Objekt der Klasse gespeichert. Die Klasse *TranslatedContent* enthält den übersetzten Inhalt einer Datei. Die Klasse *Document* stellt eine Datei der Benutzeroberfläche dar. Die Übersetzungen der Datei in verschiedene Sprachen sind die Instanzen der Klasse *TranslatedContent* auf die das Objekt der Klasse *Document* verweist.

Das folgende Objekt-Diagramm zeigt ein mögliches Szenario: drei *Document*-Objekte stellen drei Dateien der Benutzerschnittstelle da. Jede Datei hat jeweils 2 verschiedene Versionen: eine deutsche und eine englische.

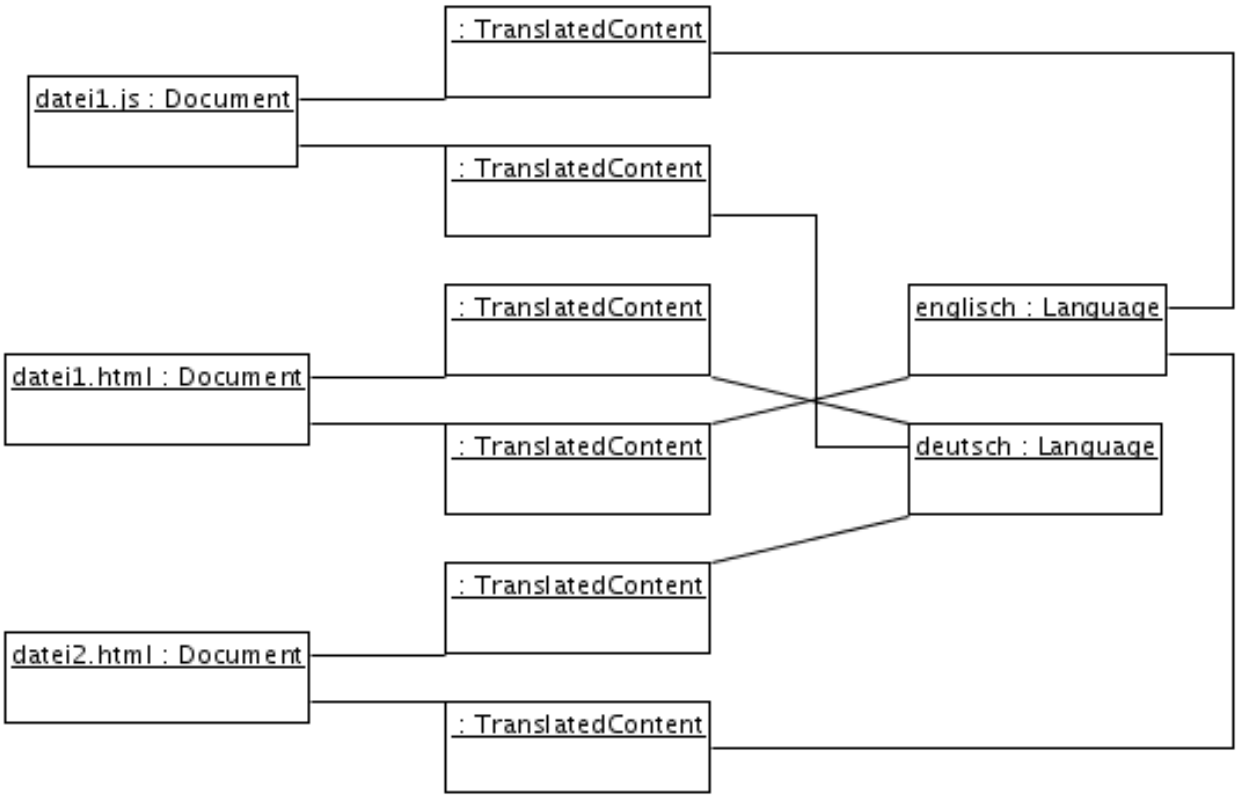


Abbildung 16: Objektdiagramm Übersetzung

Damit das Servlet weiss, welche Texte in den Dateien ersetzt werden, muss eine spezielle Formatierung für die Texte verwendet werden. Dazu wurde die zu diesem Zweck häufig verwendete Abkürzung i18n gewählt. i18n steht für das englische Wort „**I**nternationalizatio**n**“ (die 18 steht für die 18 Buchstaben zwischen i und n). Befindet sich in einer Datei der Benutzerschnittstelle der Text „i18n(Anmeldung)“ wird „Anmeldung“ durch ein entsprechendes Wort der jeweiligen Sprache ersetzt. Bei der deutschen Version wäre das das Wort „Anmeldung“, bei der englischen Version hingegen „Login“.

4.1.1.5.1 Kommunikationsprotokoll

Der Online-Compiler ist eine Client-Server-Anwendung. Client und Server tauschen Nachrichten aus. Das heißt, der Client sendet Befehle zum Server der diese dann beantwortet. Das heißt, vor der Implementierung muss festgelegt werden, wie diese Nachrichten formatiert werden. Mit anderen Worten: ein Protokoll muss festgelegt werden. Da der Online-Compiler eine Ajax-Anwendung ist, wird XML als Format für die Protokoll-Nachrichten verwendet.

Eine Client-Nachricht an das Servlet (Anfrage):

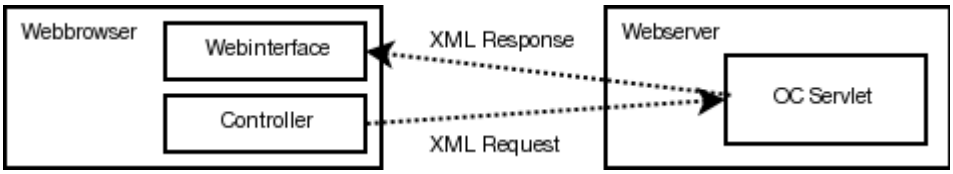
```
<oc-request>
    <befehl1 parameter1="..." parameter2="..." />
    <befehl2 parameter1="..." parameter2="..." />
    <befehl3 parameter1="..." parameter2="..." />
    …
</oc-request>
```

Das Online-Compiler-Servlet muss alle Befehle sequentiell abarbeiten. Tritt ein Fehler auf oder kann ein Befehl nicht ausgeführt werden, muss die Abarbeitung der restlichen Befehle abgebrochen werden. Der Java-Online-Compiler kann pro Anfrage nur ein Befehl ausführen. Es hat sich allerdings herausgestellt, dass es in einigen Fällen sinnvoll sein kann, mehrere Befehle auf einmal auszuführen. Zum Beispiel wenn man vor dem Compilieren einer Datei den Dateiinhalt speichern möchte. Im Java-Online-Compiler sind dazu zwei Befehle notwendig die nacheinander zum Servlet geschickt werden. Der Online-Compiler hingegen kann beide Befehle in einer Anfrage entgegennehmen.

Eine Servlet-Nachricht an den Client (Antwort):

```
<oc-response>
    <befehl1>...</befehl1>
    <befehl2>...</befehl2>
    <befehl3>...</befehl3>
    …
</oc-response>
```

Die Kommunikation zwischen Client und Server ist vergleichbar mit dem MVC-Entwurfsmuster (Modell-View-Controller):



Ein Modell, das Servlet, dient zum Speichern aller Daten (Dateien, Nutzer, Jobs usw.). Ein Controller, die Benutzerschnittstelle dient zum Eingeben von Befehlen an das Modell. Eine View, das Webinterface, dient zum Anzeigen der Modelldaten.

Im Java-Online-Compiler wurde das MVC-Entwurfsmuster dadurch implementiert, dass bei jedem Datenaustausch alle Daten des Modells (z.B. vorhandene Dateien) zum Client übertragen wurden. Dies ist zwar einfach zu Implementieren, hat aber den Nachteil, das bei jedem Datenaustausch viele Daten übertragen werden. Eine bessere Lösung ist, das bei jeder Modelländerung nur diese Änderungen zum Client übertragen werden. Modelländerungen des Servlets können Dateien betreffen und laufende Jobs.

Modelländerungen oder Modellereignisse:

- neue Datei oder neues Verzeichnis wurde erstellt
- Datei oder Verzeichnis wurde geändert
- Datei oder Verzeichnis wurde gelöscht
- Job wurde gestartet
- Job wurde beendet
- Debugger wartet (Programm welches debuggt wird, ist an einem Haltepunkt stehen geblieben)
- neue Fehlermeldungen oder Warnung eines Jobs
- Ausgabe eines laufenden Programmes

Die Ereignisse einfach zu Übertragen ist allerdings nicht ausreichend. Zum einen dürfen keine Ereignisse verloren gehen, zum anderen darf kein Ereignis mehr als einmal zum Client übertragen werden. Dieses Problem kann durch ein so genanntes Fenster gelöst werden:

Dazu muss sich das Servlet merken, welche Modellereignisse bereits gesendet wurden (**Fenster**). Der Client muss bei jeder Anfrage senden, welche Modellereignisse schon empfangen wurden. Das Servlet darf dann nur Ereignisse senden, die der Client nicht empfangen hat.

Ein weiteres Problem ergibt sich aus der Beschaffenheit des HTTP-Protokolls: weil HTTP Verbindungslos ist, nach jeder Verbindung wird die Verbindung zum Servlet wieder getrennt, weiß das Servlet nicht, zu welchem Client welche Anfrage gehört. Diese Problem wird dadurch umgangen, indem der Client (Webbrowser) bei jeder Anfrage eine eindeutige Nummer (Cookie)

zum Servlet sendet. Das Servlet kann den Client anhand dieser Nummer identifizieren.

Dieses reicht aber für den Online-Compiler nicht aus: Wenn ein Anwender mehrere Online-Compiler im gleichen Webbrowser öffnet, werden alle vom Servlet als ein Client erfasst, da alle zum Servlet die gleiche Nummer (Cookie) senden. Wenn in einem Fenster Modellereignisse empfangen werden, können die anderen Fenster diese Ereignisse nicht mehr empfangen da in dem Moment, wo ein Client Modellereignisse empfangen hat, die Ereignisse vom Servlet gelöscht werden (damit ein Ereignis nicht zweimal gesendet wird, siehe oben). Um dieses Problem zu umgehen, muss das mehrfache Öffnen von Online-Compilern in einem Webbrowser verhindert werden. Dazu werden in dem Kommunikationsprotokoll zwei Befehle benötigt: **open-session** und **close-session**. open-session muss ein Client beim Start senden. close-session muss ein Client beim Beenden senden. Wird ein open-session gesendet, nachdem ein anderer Client bereits open-session gesendet hat, wird die weitere Abarbeitung von Befehlen gestoppt. Das untere Sequenzdiagramm verdeutlicht dieses Verhalten.

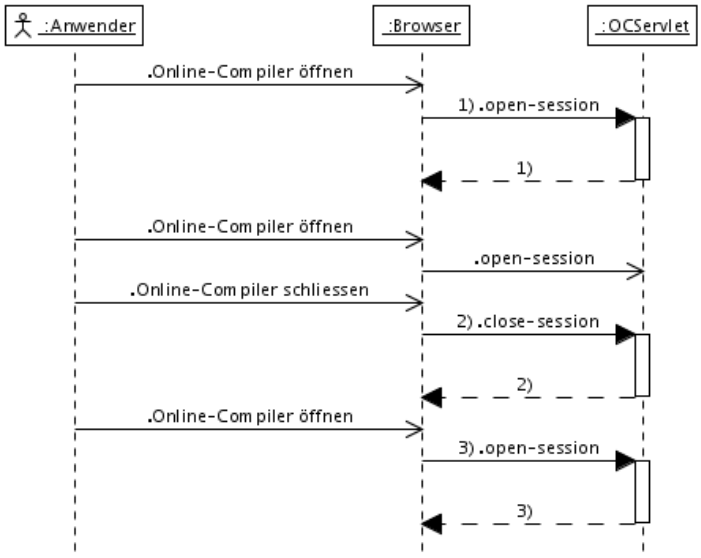


Abbildung 17: Sequenzdiagramm Kommunikation Client-Server

Wenn ein Anwender zwei Online-Compiler parallel öffnen möchte, wird der zweite Verbindungsaufbau vom Servlet unterbunden. Erst nachdem das Online-Compiler-Fenster geschlossen wird. Kann eine neue Verbindung erstellt werden.

Durch diese Lösung ergibt sich aber ein neues Problem: eine Anforderung ist eine „kleine“ Version des Online-Compilers für Ilias. Wenn ein Anwender neben dieser „kleinen“ Version auch die größere Version des Online-Compilers im gleichen Browser öffnen möchte, wird das vom Servlet unterbunden. Dieses Problem kann ebenfalls einfach gelöst werden, indem man neben der Identifikation des Browsers (Cookie) noch den Namen der Benutzeroberfläche (zum Beispiel „klein“ und „normal“ für die „kleine“ und „normale“ Version des Online-Compilers) zur Identifikation des Clients verwendet. Das Servlet identifiziert dann die Ilias-Version und die normale Version als zwei verschiedene Clients.

4.1.2 Webinterface (Client)

Das Webinterface ist die Benutzerschnittstelle des Online-Compilers. Die Benutzerschnittstelle eines Servlets wird normalerweise als Webseite, also in Form von HTML-Dokumenten, realisiert. Über diese Webseite sollen Anwender die Möglichkeit haben, Befehle zum Servlet zu senden und sich deren Ergebnisse anzuzeigen.

Dadurch, dass die Benutzerschnittstelle als Webseite in einem Web-Browser angezeigt werden soll, müssen einige Einschränkungen berücksichtigt werden, die es bei „normalen“ grafischen Benutzerschnittstellen nicht gibt. Auf diese Einschränkungen wird im Laufe dieses Kapitels näher eingegangen.

4.1.2.1 Bestandteile der Benutzerschnittstelle

Die Informationen und Daten, die in der grafischen Benutzerschnittstelle des Online-Compilers dargestellt werden wurden bereits im Kapitel „Anforderungen und Konzept“ festgelegt. Dazu zählen:

- Ein Editor mit dem Quelltextdateien von Programmen angezeigt und bearbeitet werden können.
- Eine Komponente in der Dateien und Verzeichnisse angezeigt und verwaltet (Löschen, Umbenennen, Erstellen usw.) werden können.
- Eine Komponente in der vorhandene Beispielprogramme und ggf. Übungsaufgaben angezeigt werden.
- Eine Komponente in der Fehlermeldungen und Warnungen (zum Beispiel von Compiler-Programmen) angezeigt werden.
- Eine Komponente in der die Ein- und Ausgabe von Programmen eingegeben beziehungsweise angezeigt wird.
- Eine Komponente in der Informationen zu einem laufenden Debugger-Programm angezeigt werden. Dazu zählen die aktuelle Position, in der sich das Programm befindet sowie ggf. die aktuellen Werte von Programm-Variablen.

Das Webinterface bzw. die Webseite wurde in 4 Bereiche aufgeteilt die in der folgenden Grafik dargestellt werden:

Abbildung 18: Struktur Benutzerschnittstelle

Für die „kleine“ Version des Online-Compilers, welche in das Ilias-Projekt „D+A“ eingebunden werden soll, muss die Benutzeroberfläche leicht angepasst werden unter Berücksichtigung, das der Platz für die Benutzerschnittstelle stark begrenzt ist. Die folgende Grafik zeigt die Aufteilung der Benutzerschnittstelle für die „kleine“ Version des Online-Compilers. Im Gegensatz zur „normalen“ Version wurde lediglich die Verwaltung der Dateien sowie die Anzeige der Beispiele und Übungen weggelassen, da diese Funktionen in der Ilias-Version nicht benötigt werden.

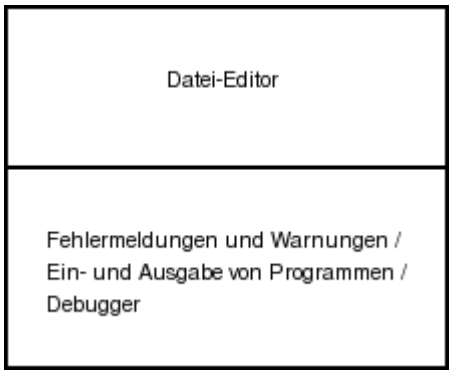


Abbildung 19: Struktur Benutzerschnittstelle Ilias-Version

Ein Problem, welches man bei dem Entwurf einer Benutzerschnittstelle berücksichtigen muss, ist der begrenzte Platz, den einem zum Darstellen zur Verfügung steht. Dieses Problem wird noch dadurch verstärkt, dass die Benutzerschnittstelle in einem Web-Browser angezeigt wird. Bei „normalen“ Desktop-Anwendungen hat der Benutzer oft verschiedene Möglichkeiten die Benutzerschnittstelle an den vorhandenen Platz anzupassen: Teile eines Fensters ausblenden oder als eigenständiges Fenster anzeigen, Teile eines Fensters in verschiedenen Registerkarten anzeigen oder teilweise sogar die Größe einzelner Elemente mit der Maus verändern. In einem Web-Browser

stehen diese Funktionen normalerweise nicht zur Verfügung.

Um das Arbeiten mit dem Online-Compiler trotzdem so einfach wie möglich zu machen, sollten einige dieser Funktionen für das Webinterface implementiert werden. Dazu zählen:

- Das Verändern der Größe von einzelnen Elementen der Benutzerschnittstelle. Dies ist zum Beispiel sinnvoll, wenn man eine große Quelltext-Datei bearbeiten möchte. In solch einem Fall kann der Benutzer einfach die Anzeige der Editor-Komponente mit der Maus vergrößern.
- Das Ein- und Ausblenden einzelner Elemente durch Verwendung von Tabs. Dadurch müssen nicht alle Informationen gleichzeitig angezeigt werden und der Anwender kann selbst bestimmen, was gerade angezeigt wird. Zum Beispiel kann man beim Ausführen eines Programms die Ein- und Ausgabe einblenden und die Anzeige vom Debugger sowie die Anzeige der Fehlermeldungen und Warnungen ausblenden.

4.1.2.2 Funktionen der Benutzerschnittstelle

Im Kapitel „Anforderungen und Ziele“ wurden die Funktionen des Online-Compilers bereits kurz erläutert. Die Benutzerschnittstelle muss diese Funktionen unterstützen. Die Funktionen im einzelnen:

- Bearbeiten von Quelltext-Dateien. Dazu soll die oben bereits erwähnte Funktion zum Anzeigen von Dateien (Editor-Komponente) dienen.
- Neben dem Bearbeiten der Dateiinhalte müssen noch Debugger-Haltepunkte (Breakpoints), Kommandozeilen-Argumente von Programmen sowie Dateien, die zu einer Programmdatei gelinkt werden sollen, sogenannte Targets, angegeben werden können
- Löschen, Umbenennen und Erstellen von Dateien und Verzeichnissen
- Öffnen von Beispielprogrammen und ggf. Übungsaufgaben
- Up- und Download von Dateien und Verzeichnissen
- Möglichkeiten zum An- und Abmelden von Benutzern
- Möglichkeiten zum Compilieren, Starten und Debuggen von Programmen
- Möglichkeiten zum Öffnen von Dokumentationen für die unterstützen Programmiersprachen und zum Durchsuchen dieser Dokumentationen

Wie bei „normalen“ Desktop-Anwendungen sollen diese Funktionen durch bekannte Komponenten wie Menüs, Popup-Menüs, Schalter und Dialogfenster realisiert werden. Die Dialogfenster sollen dabei nicht als eigenständige Fenster realisiert werden sondern als so genannter „Insite-Dialog“. Dies bedeutet, das ein Dialog nicht in einem neuen Browser-Fenster geöffnet wird (Popup-Fenster) sondern innerhalb der Webseite des Online-Compiler dargestellt wird. Die folgende Grafik zeigt einen Insite-Dialog zum Auswählen einer Datei.

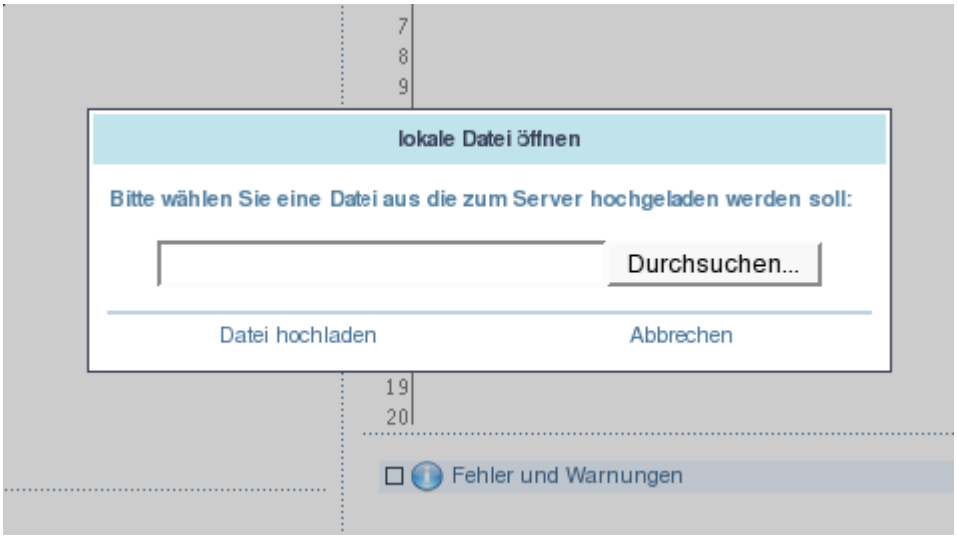


Abbildung 20: Dialog zum Auswählen einer Datei zum Hochladen

Die gleichen Funktionen könnte man auch mit normalen Popup-Fenster erreichen. Popup-Fenster werden von Anwendern aber oft als störend empfunden (Werbung) und werden aus diesem Grund oftmals auch von Browsern blockiert.

4.1.2.3 Layout

Nachdem in den ersten beiden Abschnitten auf die Inhalte und Funktionen eingegangen wurde und der Aufbau der Benutzerschnittstelle festgelegt wurde, wird in diesem Kapitel das konkrete Layout vorgestellt bzw. festgelegt.

Das Layout der Benutzerschnittstelle orientiert sich stark an dem der vorherigen Version (Java-Online-Compiler).

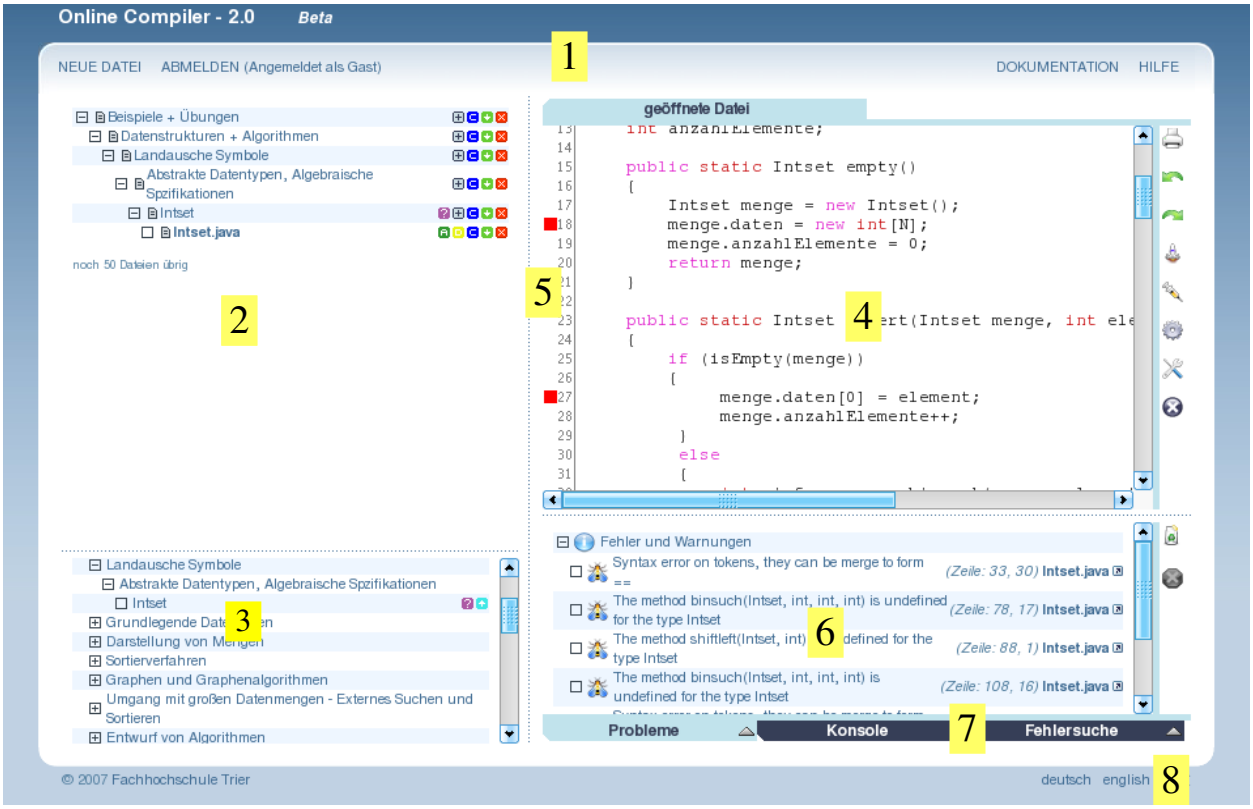


Abbildung 21: Bildschirmfoto Online Compiler

Beschreibung der einzelnen Komponenten:

1. Menü zum Erstellen von Dateien und Verzeichnissen (Menü „NEUE DATEI“), An- und Abmelden von Nutzern (Menü „ABMELDEN“), Öffnen der Programmierdokumentation (Menü „DOKUMENTATION“) und zum Öffnen der Hilfe (Menü „HILFE“)
2. Dateibaum indem alle Dateien und Verzeichnisse dargestellt werden sollen
3. Anzeige aller vorhandenen Programmierbeispiele und ggf. Übungsaufgaben
4. Editor für Quelltext-Dateien
5. Anzeige der vorhandenen Haltepunkte der geöffneten Datei
6. Anzeige für Fehlermeldungen und Warnungen
7. Tabs zum Umschalten der Anzeigen. Tabs sind: „geöffnete Datei“ in dem die Editor-Komponente enthalten ist, „Probleme“ in dem Fehlermeldungen und Warnungen angezeigt werden sollen, „Konsole“ für Ein- und Ausgabe laufender Programme sowie „Fehlersuche“ zum Anzeigen von Debugger-Informationen. Durch einen Schalter neben den Tab-Namen kann ein Tab in die obere oder untere Fensterhälfte verschoben werden. Auf diese Weise kann der Anwender selbst bestimmen, welche beiden Tabs gerade angezeigt werden sollen.
8. Schalter zum Auswählen der Sprache der Benutzeroberfläche

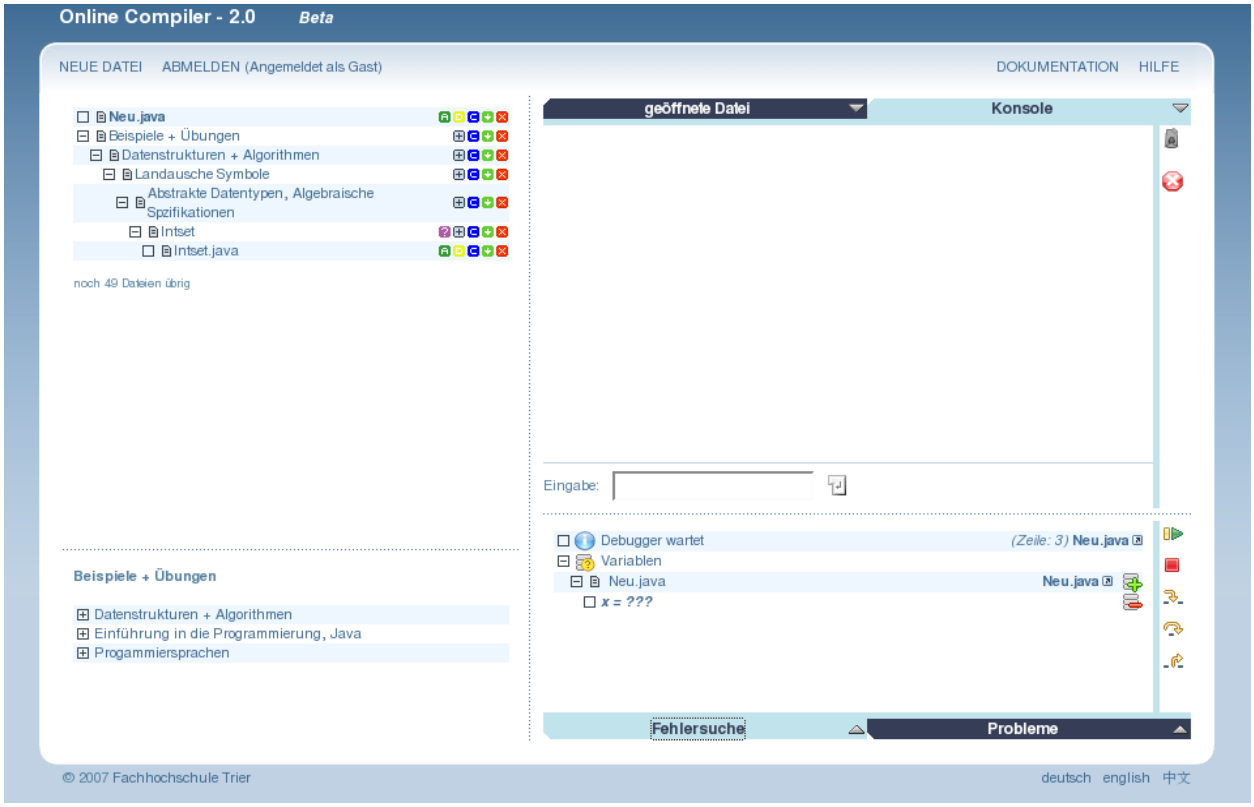


Abbildung 22: aktivierte Tabs "Konsole" und "Fehlersuche"

Wie bereits im ersten Abschnitt beschrieben, sollen die Größen der einzelnen Komponenten dynamisch mit der Maus durch die Anwender verändert werden können. Auf diese Weise kann zum Beispiel die Editor-Komponente auf fast die gesamte Bildschirm-Breite (bzw. Browser-Fenster-Breite) vergrößert werden.

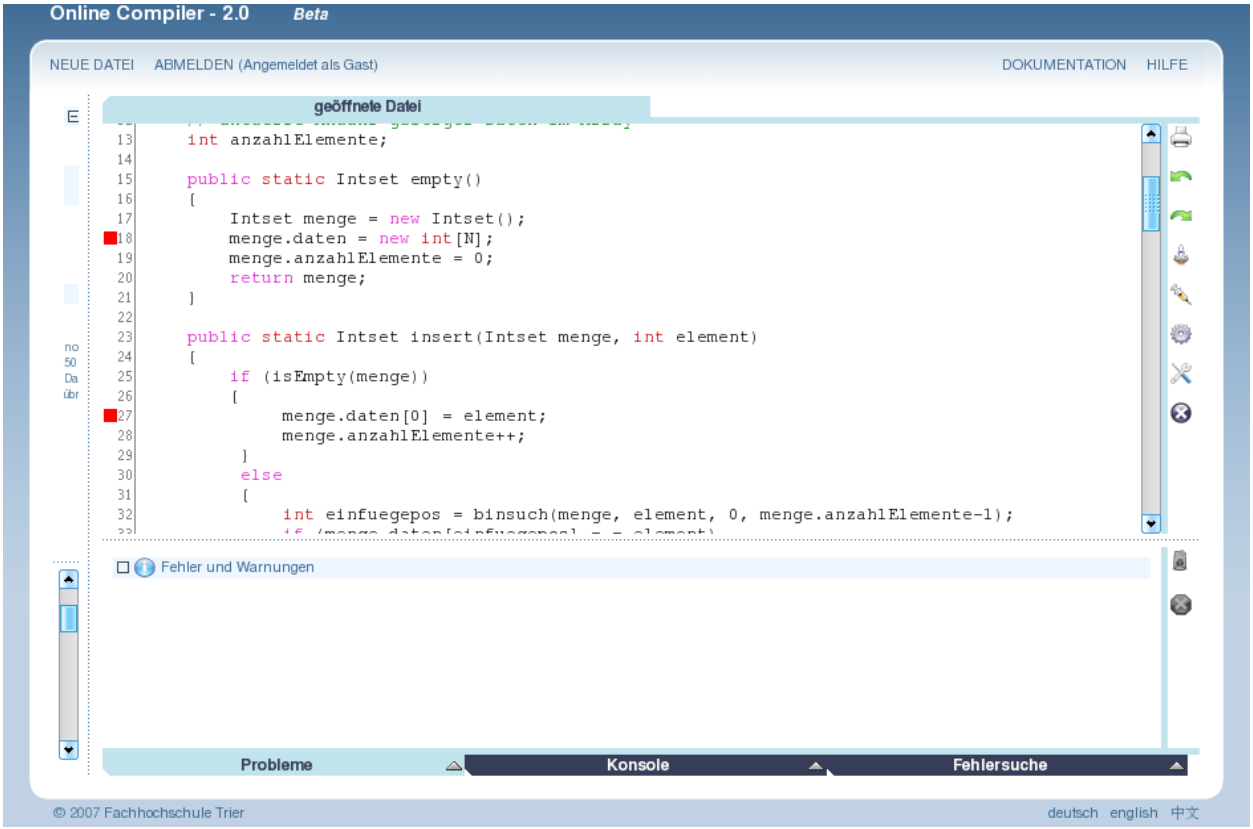


Abbildung 23: Vergrößerter Editor für eine Quelltext-Datei

Wie bereits erwähnt, muss die Ilias-Version des Online-Compilers verkleinert werden, da weniger Platz zur Darstellung zur Verfügung steht. Die Ilias-Version soll in ein Frame, welches Bestandteil eines Online-Kurses ist, eingefügt werden.

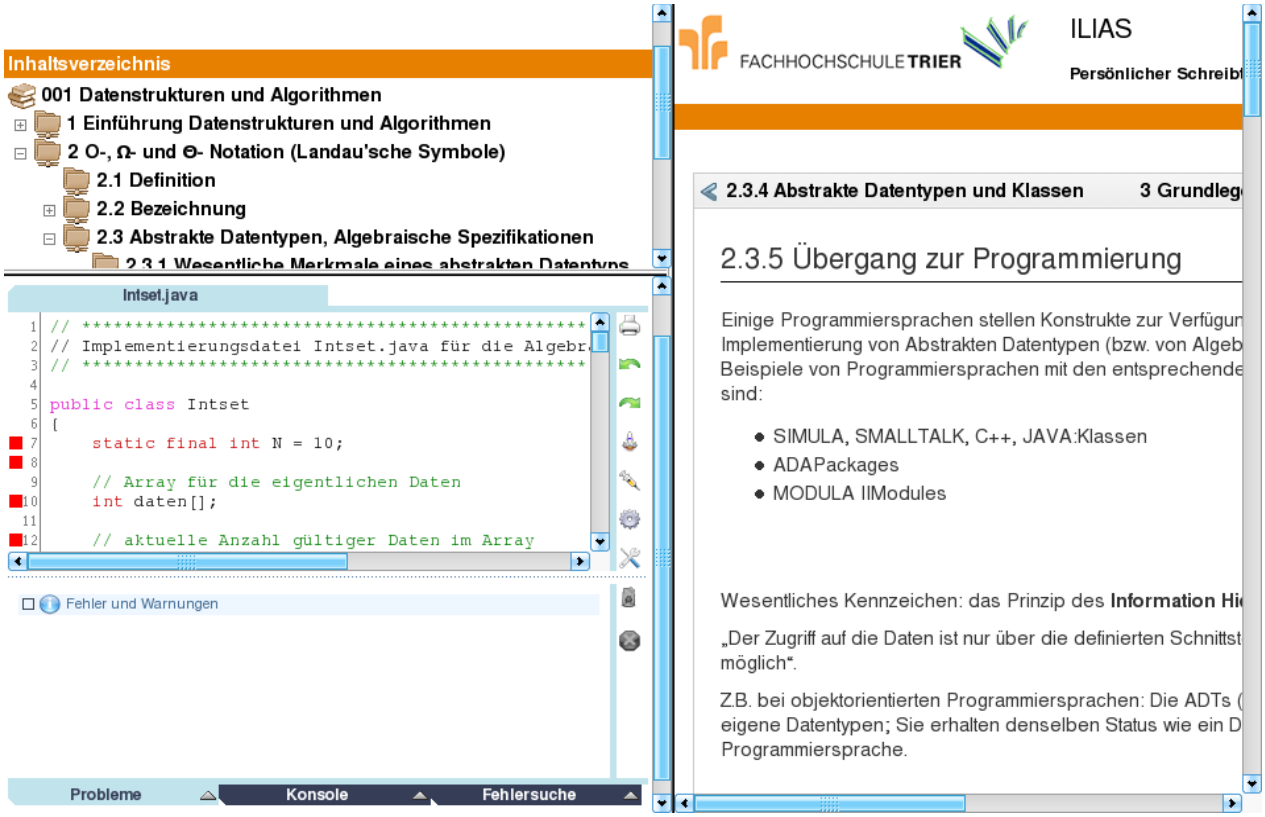


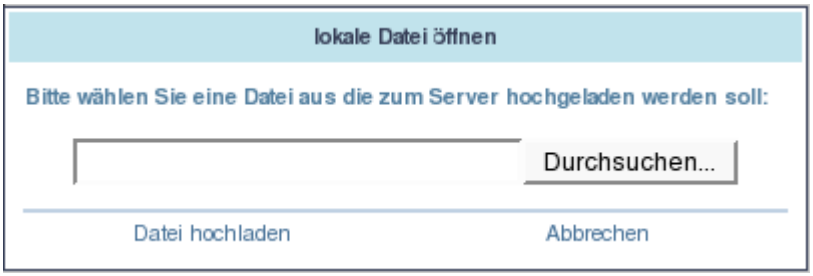
Abbildung 24: Ilias-Version des Online-Compilers integriert in Ilias

4.1.2.3.1 Dialoge

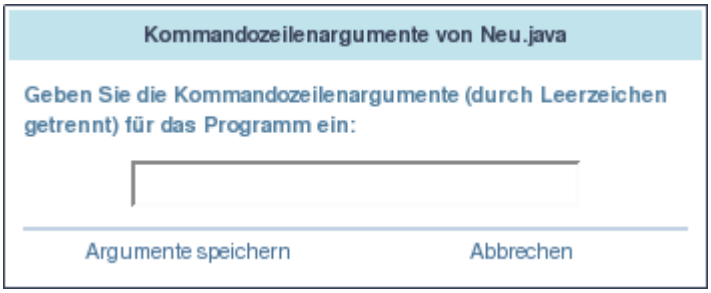
Wie bereits erwähnt, werden eine Reihe von Dialogen für den Online-Compiler benötigt. In diesem Abschnitt werden diese Dialoge kurz vorgestellt.

Dialog zum Suchen in Programmiersprachen-Dokumentationen:

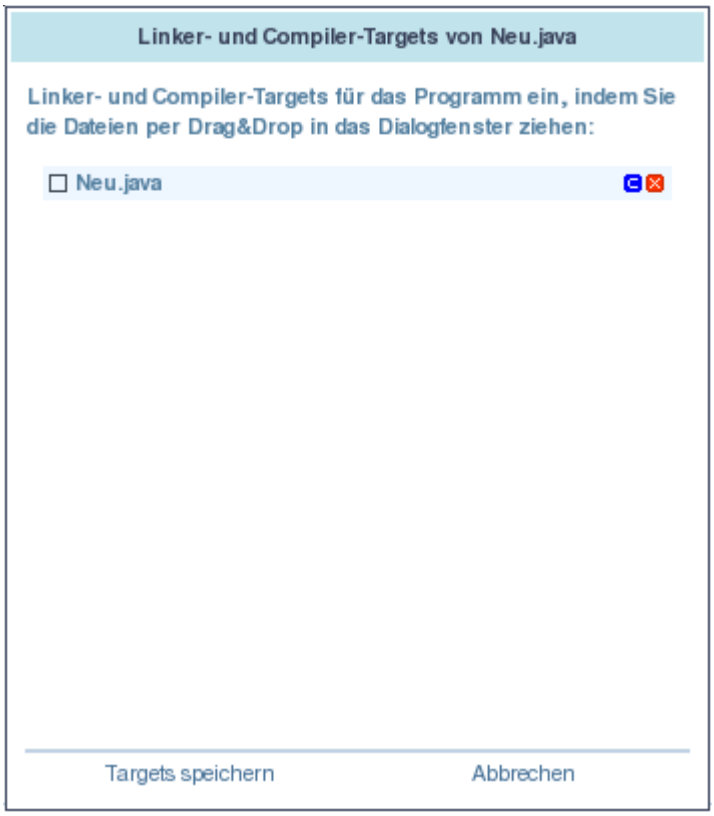
Dialog zum Auswählen einer lokalen Datei zum Hochladen:



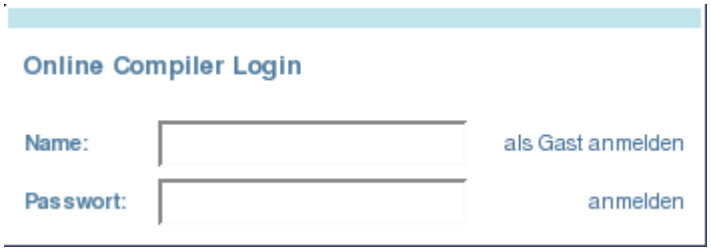
Dialog zum Eingeben von Kommandozeilenargumenten von Programmen:


Kommandozeilenargumente von Neu.java
Geben Sie die Kommandozeilenargumente (durch Leerzeichen getrennt) für das Programm ein:
Argumente speichern
Abbrechen

Dialog zum Eingeben von Linker-Targets von Programmen:


Linker- und Compiler-Targets von Neu.java
Linker- und Compiler-Targets für das Programm ein, indem Sie die Dateien per Drag&Drop in das Dialogfenster ziehen:
Neu.java
Targets speichern
Abbrechen

Dialog zum Eingeben von Benutzername und Passwort:



4.2 Multi-Task-Java

MJVM (Multi-Task-Java) soll eine Java-Anwendung sein, die es erlauben soll, verschiedenen Java-Programme parallel in einer Java-VM (Virtuall Machine) auszuführen um so die Ausführung von Java-Programmen zu beschleunigen. Die MJVM (Multi-Task-Java-VM) soll später zum Ausführen von Anwender-Programmen des Online-Compilers verwendet werden. Dazu muss es möglich sein, Programme zum Ausführen zu einer laufenden MJVM hinzuzufügen. Wenn man für jedes Java-Programm, welches ausgeführt werden soll, das MJVM-Programm neu starten müsste, wäre der Geschwindigkeitsgewinn wieder verloren. Um das Hinzufügen von Anwendungen zu einer laufenden MJVM zu ermöglichen, muss die MJVM eine Schnittstelle zur Verfügung stellen, mit der man von außerhalb auf Funktionen der MJVM zugreifen kann. Die MJVM muss also als Client-Server-Anwendung entwickelt werden. Die MJVM selbst arbeitet dabei als Server-Anwendung. Client-Anwendungen (zum Beispiel der Online-Compiler) müssen Java-Anwendungen zur MJVM hinzufügen und steuern können. So muss zum Beispiel auch die Ausgabe der einzelnen Programme vom Online-Compiler abgerufen werden können.

Die MJVM besteht im Wesentlichen aus zwei Paketen:

- *Server* für die Server-Anwendung, also die eigentliche MJVM mit der verschiedene Java-Anwendungen parallel ausgeführt werden können.
- *Client* für die Client-Anwendung beziehungsweise für die Schnittstelle mit der Anwendungen (zum Beispiel der Online-Compiler) auf eine laufende MJVM zugreifen können.

Das folgende UML-Diagramm zeigt die Klassen des Server-Paketes.

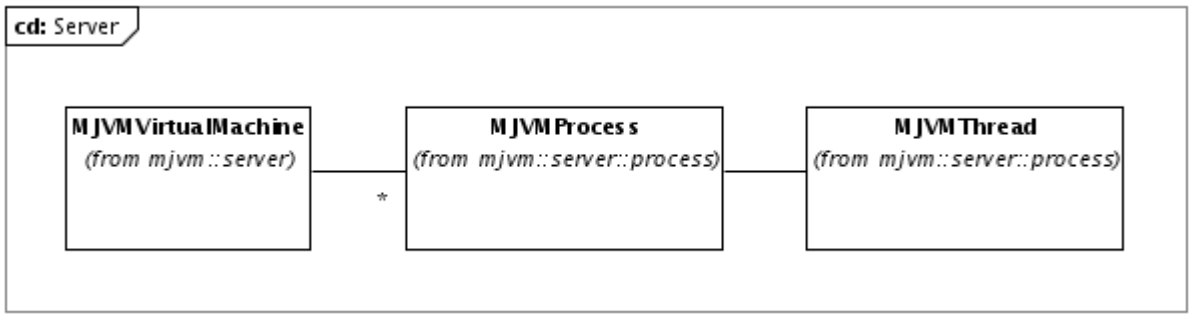


Abbildung 25: Klassendiagramm MJVM-Server

Die Klasse *MJVMVirtualMachine* enthält das eigentliche Java-Programm welches die MJVM darstellt. Objekte der Klasse *MJVMProcess* sind die Java-Programme, die in der MJVM ausgeführt werden. Da die Java-Programme parallel ausgeführt werden sollen, wird für jedes Java-Programm ein eigener Thread benötigt: die Klasse *MJVMThread*.

Das nächste Klassendiagramm zeigt die Klassen, die von einer Client-Anwendung (zum Beispiel dem Online-Compiler) zum Zugriff auf eine MJVM benötigt werden:

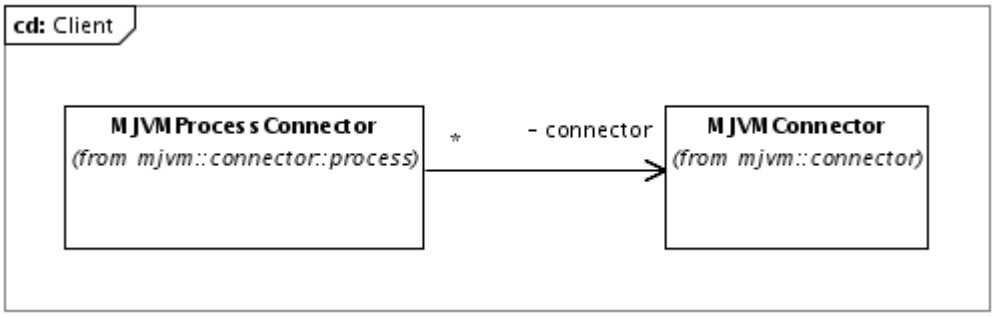


Abbildung 26: Klassendiagramm MJVM-Client

Die Klasse *MJVMProcessConnector* ist die Schnittstelle zu einer MJVM-Anwendung. Die Klasse *MJVMConnector* ist die Schnittstelle zu einer einzelnen Java-Anwendung, welche in einer MJVM-Anwendung ausgeführt wird.

5 Implementierung

Nachdem, wie im vorherigen Kapitel beschrieben, das Design festgelegt und ein Entwurf erstellt wurde, erfolgt die Implementierung der Anwendungen wie im folgenden Abschnitt beschrieben.

Da das komplette System Online-Compiler aus zwei Anwendungen, einer Client- und einer Serveranwendung bzw. einem Webinterface und einem Servlet besteht, werden die Implementierungen dieser beiden Anwendungen jeweils in einem eigenen Abschnitt behandelt.

Der dritte Abschnitt befasst sich mit der Implementierung der Multi-Task-Java-Anwendung (MJVM).

Da die gesamte Anwendung mehr als 23000 Zeilen Quellcode (ohne Kommentare und Leerzeilen) enthält, werden aus Platzgründen hier nur die wichtigsten und entscheidenden Elemente der Implementierung behandelt.

Eine ausführliche Dokumentation befindet sich ebenfalls in den Quelltexten im CD-Verzeichnis `source` sowie im Verzeichnis `doc/api` (Javadoc).

5.1 Online-Compiler

Wie bereits oben erwähnt, besteht der Online-Compiler aus zwei separaten Anwendungen: dem Client und dem Servlet. Beide werden auch getrennt voneinander betrachtet. Der erste Abschnitt beschäftigt sich mit der Implementierung des Servlets während im zweiten Abschnitt auf die Implementierung der Benutzerschnittstelle, dem Client, eingegangen wird.

5.1.1 Das Servlet

Ziel dieses Kapitels ist die Beschreibung von Implementierungsdetails des Servlets Online-Compiler. Das Servlet ist eine Java Anwendung, die von einem sogenannten Servlet-Container, dem Tomcat-Webserver, ausgeführt wird.

Neben diesem Kapitel steht auch noch eine ausführliche **Inline-Dokumentation** in den Java Quelldateien (Javadoc) zur Verfügung. Diese befindet sich im Verzeichnis `doc/api/OnlineCompiler` auf der CD oder kann Online über die URL http://javock.fh-trier.de:8080/oc/api/index.html abgerufen werden.

5.1.1.1 Datei-Übersicht

Die Quelldateien des Servlets befinden sich im Verzeichnis `source/OnlineCompiler/WEB-INF/classes`. Das Servlet besteht aus folgenden Java-Paketen:

Paketname	*Inhalt*
onlinecompiler.exceptions	Dieses Paket enthält Ausnahmeklassen des Online-Compilers.
onlinecompiler.ilias	In diesem Paket befindet sich die Anbindung zum Ilias-Server.
onlinecompiler.languages	Dieses Paket enthält die Unterstützung für die verschiedenen Programmiersprachen.
onlinecompiler.os.security	Klassen, die zum Schutz des Servers vor den Benutzer-Programmen dienen, befinden sich in diesem Paket.
onlinecompiler.servlet	Dieses Paket enthält das eigentliche Servlet.
onlinecompiler.source	In diesem Paket befinden sich alle Klassen zum Verwalten von Dateien der Benutzer.
onlinecompiler.test	Dieses Paket enthält ein Programm zum durchführen von Testfällen (siehe Kapitel „Test“).
onlinecompiler.tools	Dieses Paket enthält einige allgemeine Klassen mit Hilfsfunktionen.
onlinecompiler.user	In diesem Paket sind alle Klassen enthalten, die für die Benutzerverwaltung nötig sind.

Neben den Quelltext-Dateien des Online-Compiler-Servlets werden noch eine Reihe weitere Programm-Bibliotheken verwendet. Die meisten werden für die XML-Unterstützung benötigt.

Benötigte Programm-Bibliotheken des Online-Compiler-Servlets:

Name	***Beschreibung***	***benötigte Version***
Apache XML Resolver	Bestandteil von Apache Xalan	1.2 oder höher
Apache XML Serializer	Bestandteil von Apache Xalan	2.7 oder höher
Apache Xalan	Bibliothek zum Übersetzen von XML-Dateien in andere Formate	2.7 oder höher
Apache Xerces	XML Parser	2.7.1 oder höher
Apache Xerces Samples	Bestandteil von Apache Xerces	2.7.1 oder höher
Apache XML Apis	Schnittstelle für XML Standards	2006 oder jünger
SUN Servlet	Sun Java Servlet Package	2.5 oder höher
Apache Commons Fileupload	Ermöglicht Dateiupload mit Servlets	1.1.1 oder höher
Apache Commons IO	Datei-Bibliothek für Servlets	1.3.1 oder höher
Eclipse JDT	Java Development Tools	3.2.2 oder höher

Die Programmbibliotheken, die vom Online-Compiler-Servlet benötigt werden, befinden sich im Verzeichnis `source/OnlineCompiler/javalibs` auf der CD.

5.1.1.2 Compilierung des Servlets

Das Servlet ist vollständig in Java Programmiert worden. Um es zu Compilieren, müssen alle Quelltextdateien im Verzeichnis `source/OnlineCompiler/WEB-INF/classes` compiliert werden. Damit das Servlet compiliert werden kann, sollte darauf geachtet werden, dass alle benötigten Programmbibliotheken zur Verfügung stehen (siehe vorherigen Abschnitt).

5.1.1.3 Implementierung des Servlets

5.1.1.3.1 Dateien

Die Quelltextdateien müssen von den Backend-Programmen (Compiler und Linker) aufgerufen werden können. Dazu müssen die Dateien auf der Festplatte des Servers gespeichert werden. Jeder angemeldete Anwender des Online-Compiler bekommt dafür ein eigenes Verzeichnis in dem alle Dateien, die er erstellt hat, gespeichert werden.

Die Verzeichnisse der Anwender werden im Unterverzeichnis *users* des Online-Compiler-Verzeichnisses (zum Beispiel */srv/oc*) erstellt:

- das Unterverzeichnis *users/admin* enthält die Verzeichnisse der Administratoren
- das Unterverzeichnis *users/ilias* enthält die Verzeichnisse der Ilias-User
- das Unterverzeichnis *users/tempusers* enthält die Verzeichnisse der Gäste

Das Speichern der Dateien erfolgt bei Änderung des Dateiinhaltes. Das heißt, jedesmal wenn ein Anwender eine Datei bearbeitet hat und der Client den neuen Dateiinhalt zum Online-Compiler-Servlet sendet, wird die Datei auf der Festplatte gespeichert. Dazu dient die Methode *setContent* der Klasse *TextFile*.

```
public void setContent(String content)
     throws OnlineCompilerIOException
{
     try{
          if(content.length() > Server.getMaxFileSize())
               content = content.substring(0,
                    Server.getMaxFileSize());
          FileTools.createDirectories(getAbsoluteDirectory());
          FileTools.saveTextFile(this.getAbsoluteFileName(),
               content);
          contentChanged(content);
     }catch(IOException e){
          throw new OnlineCompilerIOException();
     }
}
```

Wenn der Inhalt einer Datei geöffnet wird, wird die Datei wieder von der Festplatte gelesen. Das heißt, der Dateiinhalt wird nicht permanent im Arbeitsspeicher zwischengespeichert.

5.1.1.3.2 Ilias-Anbindung

Eine wichtige Anforderung ist die Benutzer-Authentifizierung über den Ilias-Server. Das bedeutet, dass sich Anwender mit ihrem Ilias-Benutzernamen und ihrem Ilias-Passwort an den Online-Compiler anmelden können. Dazu muss der Online-Compiler Zugriff auf einen Ilias-Server haben.

Ilias besitzt eine sogenannte SOAP-Schnittstelle. SOAP ist ein Protokoll, mit dem RPCs (Remote Procedure Calls) ausgeführt werden können. Remote Procedure Call bedeutet, dass eine Anwendung (die Client-Anwendung) einen Methodenaufruf an eine andere Anwendung (die Server-Anwendung) sendet. Die Methode wird von der Server-Anwendung ausgeführt und das Ergebnis an die Client-Anwendung zurückgeschickt. Client- und Server-Anwendung können auf verschiedenen Rechnern laufen. Die Übertragung der SOAP-Nachrichten erfolgt in der Regel über HTTP. Ilias hat eine Reihe von Funktionen, die eine Client-Anwendung mit SOAP aufrufen kann. Die komplette Liste der Ilias-SOAP-Funktionen kann man über https://ilias.fh-trier.de/webservice/soap/server.php abrufen. Um eine Benutzerauthentifizierung über Ilias durchzuführen, kann die SOAP-Funktion *login* verwendet werden. Diese Funktion hat drei Parameter: den Namen des Ilias-Service, den Namen des Anwenders und das Passwort des Anwenders. Wenn auf dem Ilias-Server ein Nutzer mit dem Namen existiert, wird als Ergebnis eine eindeutige Nutzerkennung zurückgesendet. Wenn kein Nutzer mit dem Namen und dem Passwort existiert, wird eine Fehlermeldung generiert. Auf diese Weise kann man mit Ilias eine Benutzer-Authentifizierung durchführen.

Die Ilias-Anbindung wird von der Klasse *IliasServer* durchgeführt. In dieser Klasse sind zwei Methoden vorhanden: *init* und *checkLogin*. Die Methode init initialisiert die Verbindung zu dem Ilias-Server und liest Konfigurations-Daten ein. Dazu zählen die URL der SOAP-Schnittstelle und der Name des Ilias-Service:

```
public static void init()
{
      // Konfiguration laden
      Server.serverConfig().createProperty("ilias.server.url",
            "http://localhost/ilias3/webservice/soap/server.php");
      Server.serverConfig().createProperty("ilias.server.client",
            "abschlussarbeit");
      serverURL =
            Server.serverConfig().getString("ilias.server.url");
      clientName =
            Server.serverConfig().getString("ilias.server.client");
      try{
            service = new Service();
            call = (Call)service.createCall();
      }catch(Exception e){
            Server.errorMessage(e);
      }
}
```

Die zweite Methode *checkLogin* führt die eigentliche Benutzer-Authentifizierung durch. Dazu wird über SOAP die Methode *login* aufgerufen:

```
public synchronized static boolean checkLogin(String username,
     String password)
{
     try{
          call.setTargetEndpointAddress( new
               java.net.URL(serverURL));
          String res = (String) call.invoke("login", new Object[]
               {clientName, username, (password == null ? "" :
               password)} );
          call.invoke("logout", new Object[] {res} );
          return res.endsWith("::" + clientName);
     }catch(Exception e){
          Server.errorMessage(e);
          return false;
     }
}
```

5.1.1.3.3 C und C++

Wichtigste Aufgabe des Servlets ist natürlich das Erstellen, Ausführen und Debuggen von Programmen in den verschiedenen Programmiersprachen.

5.1.1.3.3.1 Compilieren

Um C- und C++-Programme zu erzeugen, wird GCC, die GNU-Compiler-Collection, verwendet. Zum Erzeugen von C-Programmen gibt es das Kommandozeilen-Programm gcc. Zum Erzeugen von C++-Programmen wird das Kommandozeilen-Programm g++ verwendet. Die Klasse *CFileBuildJobGNU* dient als Schnittstelle zu den Kommandozeilen-Programmen gcc und g++. Wie bereits im Entwurf festgelegt wurde, ist die Klasse *CFileBuildJobGNU* (bzw. deren Instanzen) ein so genannter Job. Also ein Thread, der bestimmte Aufgaben, in diesem Fall das Erzeugen von C- und C++-Programmen, durchführt.

Um C- und C++-Programme zu erzeugen, werden zwei Operationen benötigt:

- Erzeugen von sogenannten Objekt-Dateien durch Compilieren der Quelltextdateien.
- Erzeugen des ausführbaren Programmes durch das sogenannte Linken: mehrere Objekt-Dateien werden zu einem Programm (Binary) zusammengefasst.

Für diese beiden Operationen wurden zwei Methoden in der Klasse *CFileBuildJobGNU* erstellt: *compileFile* zum Compilieren einzelner Dateien und *buildFile* zum Linken einzelner Dateien.

```
protected boolean compileFile()
     throws IOException, InterruptedException,
          OnlineCompilerException
{
     setProcess(new String[]{(cfile instanceof CPPFile ? "g++" :
          "gcc"), "-g", "-c", cfile.getAbsoluteFileName()},
          cfile.getAbsoluteDirectory(), CLanguage.processTimeout,
          CLanguage.ioCharset);
     getProcess().waitFor();
     if(isProcessTimeout(true))
          throw new TimeoutException();
     parseGCCResult();
     if(getProcess().exitValue() != 0)
          throw new OnlineCompilerException("Compiler failed");
     else {
          cfile.setRecompile();
          return true;
     }
}
```

Die Methode *compileFile* ruft zunächst dass Programm gcc für C-Dateien bzw. g++ für C++-Dateien auf. Das Objekt *cfile* ist die Datei, die compiliert werden soll. Nach dem Starten des Prozesses wird mit der Methode *waitFor* gewartet, bis die Compilierung der Datei beendet ist. Mit der Methode *parseGCCResult* werden eventuelle Fehlermeldungen des Compilers analysiert und zum Client gesendet.

Um ein ausführbares Programm zu erzeugen, muss nach dem Compilieren noch der sogenannte Linker aufgerufen werden. Wie bereits oben erwähnt, wird dazu die Methode *buildFile* verwendet.

```
protected boolean buildFile()
      throws IOException, InterruptedException,
            OnlineCompilerException
{
      Vector<CFile> targets = new Vector<CFile>();
      for(int i = 0; i < cfile.getTargetCount(); i++)
            if(cfile.getTarget(i).getLink())
                  targets.add((CFile)cfile.getTarget(i).getTarget());
      String[] cmd = new String[3 + targets.size()];
      cmd[0] = (cfile instanceof CPPFile ? "g++" : "gcc");
      for(int i = 0; i < targets.size(); i++)
            cmd[i + 1] = targets.get(i).getAbsoluteDirectory() +
                  FileTools.sep +
                  targets.get(i).getTargetLinkFileName();
      cmd[cmd.length - 2] = "-o"; cmd[cmd.length - 1] =
            cfile.getBinaryName();
      setProcess(cmd, cfile.getAbsoluteDirectory(),
            CLanguage.processTimeout, CLanguage.ioCharset);
      getProcess().waitFor();
      if(isProcessTimeout(true))
            throw new TimeoutException();
      parseGCCResult();
      if(getProcess().exitValue() != 0)
            throw new OnlineCompilerException("Linker failed");
      else {
            cfile.setRebuild();
            cfile.sandbox();
            return true;
      }
}
```

Die Funktion *buildFile* ähnelt der Funktion *compileFile*: zuerst wird ein Prozess, der Linker, gestartet, dann auf dessen Ende gewartet und zum Schluss Fehlermeldungen mit der Funktion *parseGCCResult* zum Client gesendet. Der Hauptunterschied zwischen beiden Funktionen ist, dass bei *buildFile* das Linker-Programm gcc bzw. g++ mit den Dateien, die gelinkt werden sollen, aufgerufen werden muss. Die Dateien, die der Linker zu einem Programm zusammenfügen soll, sind die Target-Dateien der Datei *cfile*. Ein weiterer Unterschied ist, dass nach dem Erzeugen des Programms mit der Funktion *sandbox* eine AppArmor-Sandbox für das erzeugt Programm erstellt

wird. Nähere Informationen zu AppArmor befinden sich im Abschnitt „AppArmor“.

5.1.1.3.3.2 Ausführen

Das Ausführen von C- und C++-Programmen wird von der Klasse *CFileLaunchJobGNU* durchgeführt. Wie die Klasse zum Erstellen von C- und C++-Programmen ist auch diese Klasse von der Klasse *Job* abgeleitet. Dass heißt, das Ausführen von C- und C++-Programmen wird ebenfalls von einem eigenen Thread durchgeführt. Einzige Methode der Klasse ist *launchFile*, welche von dem Job-Thread ausgeführt wird:

```
protected boolean launchFile()
      throws IOException, InterruptedException,
            OnlineCompilerException
{
      cfile.sandbox(); // Sandbox ggf. erstellen
      String[] cmd =
            cfile.getCommandLineArgs(cfile.getAbsoluteBinaryName());
      setProcess(cmd, cfile.getAbsoluteDirectory(),
      CLanguage.processTimeout, CLanguage.ioCharset);
      getProcess().waitFor();
      if(CLanguage.appArmor && getProcess().exitValue() == 255)
            throw new SandboxException();
      if(isProcessTimeout(true))
            throw new TimeoutException();
      return true;
}
```

LaunchFile startet das C- oder C++-Programm und wartet, bis das Programm beendet ist. Eine Besonderheit von C- und C++-Programmen ist der Exit-Code, also die Zahl die von dem Programm als Ergebnis-Wert zurückgegeben wird. Falls das Programm mit einer AppArmor-Sandbox ausgeführt wird, ist der Exit-Code immer dann 255, wenn ein unerlaubter Befehl ausgeführt wurde. Dass heißt, wird zum Beispiel versucht in eine Datei zu schreiben, wird das Programm von AppArmor beendet und als Exit-Code 255 zurückgeliefert.

5.1.1.3.3.3 Debuggen

Der bei weitem komplexeste Teil der C- und C++-Implementierung des Online-Compilers ist der Debugger. Das Debuggen von C- und C++-Programmen wird mit Hilfe des Kommandozeilen-Programms GDB (GNU Debugger *[FSF 2]*) durchgeführt. Der GDB ist ein Programm, welches ein C- oder C++-Programm debuggen kann. Gesteuert wird der GDB mit Hilfe von Befehlen, die normalerweise über die Kommandozeile eingegeben werden.

Die Klasse *CFileDebugJobGNU* führt das Debuggen von C- und C++-Programmen aus. Wie die beiden vorher genannten Klassen, *CFileLaunchJob* und *CFileBuildJob* ist auch diese Klasse von der Klasse *Job* abgeleitet. Dass heißt, ein eigener Thread führt das Debuggen von C- und C++-

Programmen aus.

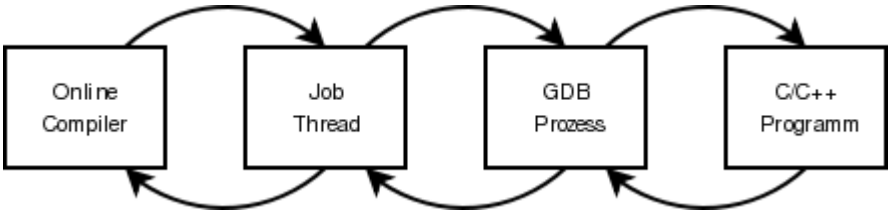


Abbildung 27: Threads und Prozesse zum Debuggen von C- und C++-Programmen

Bei der Implementierung des Debuggers muss berücksichtigt werden, dass der GDB nicht immer Befehle entgegennehmen kann. Der GDB verarbeitet nur dann Befehle, wenn das Programm, welches debuggt werden soll, nicht aktiv ist. Also genau dann, wenn das Programm noch nicht gestartet wurde oder wenn das Programm gestoppt wurde (zum Beispiel weil ein Debugger-Haltepunkt erreicht wurde). Der GDB hat also zwei verschiedene Zustände: „Programm läuft“ wenn das Programm gerade aktiv ist und „Debugger wartet“, wenn das Programm gestoppt wurde. Diese Zustände müssen auch bei der Implementierung der Klasse *CFileDebugJobGNU* berücksichtigt werden. Das folgende Zustandsdiagramm zeigt die Zustände der Klasse *CFileDebugJobGNU*. Neben den beiden Zuständen „Programm läuft“ und „Debugger wartet“ werden noch ein Zustand zum Starten des GDB und des Programms sowie ein Zustand zum Beenden des GDB und des Programms benötigt.

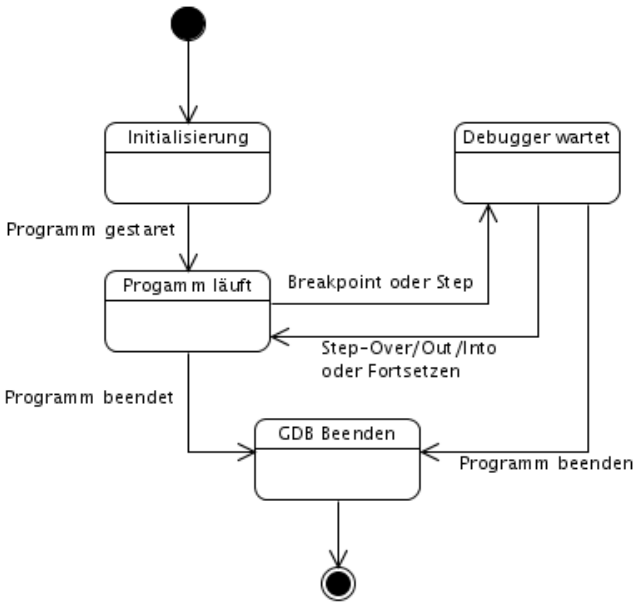


Abbildung 28: Zustände des C/C++-Debugger-Jobs

Die Methode *setDebuggerWaiting* dient zum Wechseln zwischen den Zuständen „Programm läuft“ und „Debugger wartet“. Wenn das Programm gestoppt wurde (zum Beispiel weil ein Debugger-Haltepunkt erreicht wurde), wird die Methode *setDebuggerWaiting* mit dem Parameter *true* aufgerufen. Dadurch wird in den Zustand „Debugger wartet“ gewechselt, woraufhin Befehle (zum Beispiel „Step-Into“) eingegeben werden können.

```
private synchronized void setDebuggerWaiting(boolean
     debuggerWaiting)
{
     this.debuggerWaiting = debuggerWaiting;
}
```

Im Zustand „Debugger wartet“ muss auf Befehle gewartet werden. Dazu muss der Job-Thread gestoppt werden wozu die *wait*-Methode des Threads verwendet wird. Nachdem ein Befehl eingegeben wurde, wird der Job-Thread mit der *notify*-Methode wieder gestartet und danach in den Zustand „Programm läuft“ oder „GDB beenden“ gewechselt.

```
public synchronized void stepOver()
     throws OnlineCompilerException
{
     if(!isDebuggerReady())
          throw new OnlineCompilerException("Debugger not
                                            Running");
     setDebuggerWaiting(false);
     gdbCommand("next");
     restartWaitThread();
}
```

Die Implementierung der Methode *stepOver* der Klasse *CFileDebugJobGNU* führt den Befehl „Step-Over“ aus. Mit diesem Befehl wird genau eine Programmzeile vom C- oder C++-Programm ausgeführt. Am Methodenanfang wird zunächst geprüft, ob der Debugger sich im Zustand „Debugger wartet“ befindet. Wenn dies nicht der Fall ist, wird eine Fehlermeldung erzeugt. Ansonsten wird mit der Methode *gdbCommand* der GDB-Befehl *next* zum GDB gesendet. Daraufhin führt der GDB den „Step-Over“ Befehl aus. Mit der Methode *restartWaitThread* wird schließlich der Job-Thread wieder gestartet und damit in den Zustand „Programm läuft“ gewechselt.

Das eigentliche Debuggen wird von der Methode *debugFile* der Klasse *CFileDebugJobGNU* durchgeführt. In der Methode *debugFile* wird zunächst der GDB mit dem C- bzw. C++-Programm gestartet:

```
setProcess(new String[]{"gdb", "--return-child-result", "--
     fullname", "--quiet", "--nx", cfile.getAbsoluteBinaryName()},
cfile.getAbsoluteDirectory(), CLanguage.processTimeout,
CLanguage.ioCharset, outputFile, errorFile);
getWatchdog().pauseWatchdog();
...
```

```
setDebuggerWaiting(true);
getCommandResult(); // Init-Meldung lesen
gdbCommand("set confirm off");
getCommandResult();
pid = getProcessID(true);
for(int i = 0; i < cfile.getTargetCount(); i++)
     updateBreakpoints(cfile.getTarget(i).getTarget());
gdbCommand("run " + cfile.getArgs() + " 1>\"" + outputFile + "\"
     2>\"" +   errorFile + "\"");
setDebuggerWaiting(false);
```

Mit der Methode *updateBreakpoints* werden alle Debugger-Haltepunkte zu dem GDB hinzugefügt. Alle vorhandenen Debugger-Haltepunkte müssen vor dem Start des C- oder C++-Programmes dem GDB mitgeteilt werden.

Der GDB-Befehl *run* startet das zu debuggende C- oder C++-Programm. Mit *setDebuggerWaiting* wird nach dem Start des Programmes in den Zustand „Programm läuft“ gewechselt und die folgende While-Schleife ausgeführt:

```
while(true){
     ...
     boolean newPosition = parseResultForSuspend();
     ...
     if(isProcessTimeout() || gdbKilled || isStopped()){
          ...
          return false;
     }
     synchronized(this){
          setDebuggerWaiting(true);
          for(int i = 0; i < updateBreakpoints.size(); i++)
               updateBreakpoints(updateBreakpoints.get(i));
          updateBreakpoints.clear();
     }
     fireJobWaiting(new DebugJobWaitingResult(lastFile, lastLine));
     waitForRunning();
}
```

Am Anfang der Schleife wird die Methode *parseResultForSuspend* aufgerufen. Diese Methode liest solange den Ausgabe-Strom des GDB-Prozesses, bis der GDB eine Ausgabe erzeugt hat. Der GDB gibt, falls ein Befehl ausgeführt wurde oder falls das C- oder C++-Programm gestoppt wurde eine entsprechende Status-Meldung aus die mit einem „(GDB)“ abgeschlossen wird. Dass heißt, solange kein „(GDB)“ im Ausgabe-Strom des GDB-Prozesses enthalten ist, ist das C- oder C-++Programm nicht gestoppt oder der GDB führt gerade einen Befehl aus. Die Methode *parseResultForSuspend* wartet also solange, bis der GDB entweder im Zustand „Debugger wartet“ ist oder das C- oder C++-Programm beendet ist (Zustand „Programm beendet“). Mit der folgenden if-Anweisung wird überprüft, ob der aktuelle Debugger Zustand „Programm beendet“ ist. Wenn dies der Fall ist, wird die Schleife und das Debuggen beendet. Falls der Zustand „Debugger wartet“ ist, werden zunächst

die Debugger-Haltepunkte aktualisiert und danach mit der Methode *fireJobWaiting* eine Meldung zum Client gesendet.

Im vorherigen Abschnitt wurden bereits einige Befehle beschrieben, die vom Online-Compiler verwendet werden, um den GDB und das zu debuggende Programm zu steuern. In der folgenden Tabelle werden alle verwendeten GDB-Befehle aufgelistet.

GDB-Befehl	***Beschreibung***	***wird verwendet in(Methoden)***
run	Startet die Ausführung des C- oder C++-Programms	*debugFile*
step	Führt ein „Step-Into“ aus, springt in eine Methode	*stepInto*
next	Führt ein „Step-Out“ oder „Step-Over“ aus, springt aus einer Methode oder führt einen Befehl aus	*stepOut, stepOver*
continue	Setzt die Ausführung eines Programms fort	*resume*
break	erstellt einen Debugger-Haltepunkt	*addBreakpoint*
delete breakpoints	löscht einen Debugger-Haltepunkt	*removeBreakpoint*
print	liest den Wert einer Variablen	*updateVariable*
set confirm off	deaktiviert die Eingabe von y (für yes) und n (für no) für die Bestätigung von Befehlen	*debugFile*

5.1.1.3.4 Java

5.1.1.3.4.1 Compilieren

Das Compilieren bzw. Erzeugen von Java-Programmen ähnelt stark dem Erzeugen von C- und C++-Programmen. Der größte Unterschied ist, das Java-Programme, im Gegensatz zu C- und C++-Programmen, nicht gelinkt werden müssen. Dass heißt, um Java-Programme zu Erstellen, reicht es aus, die Java-Quelltext-Dateien zu compilieren. Java-Programme werden, im Gegensatz zu C- und C++-Programmen, nicht mit einem externen Programm wie gcc oder g++ compiliert. Um Java-Dateien zu compilieren, wird der Eclipse-Java-Compiler *[JDT]* verwendet. Dieser kann, im Gegensatz zum SUN Java-Compiler, innerhalb einer laufenden VM (Virtuall Machine) ausgeführt werden. Dass heißt, man muss nicht einen neuen Prozess zum Compilieren einer Datei starten.

Die Klasse Java*FileBuildJobGNU* dient als Schnittstelle zu dem Eclipse-Java-Compiler. Wie bereits im Entwurf festgelegt wurde, ist die Klasse *JavaFileBuildJobGNU* (bzw. deren Instanzen) ein so genannter Job. Also ein Thread, der bestimmte Aufgaben, in diesem Fall das Erzeugen von Java-Programmen, durchführt.
Da Java-Dateien nicht compiliert werden müssen, hat die Methode *buildFile* zum Linken einzelner Dateien keine Funktion. Die Methode *compileFile* dient, wie schon bei der Klasse *CFileBuildJobGNU*, zum Compilieren:

```
protected boolean compileFile()
      throws IOException, InterruptedException,
            OnlineCompilerException
{
      String cmd = "-noExit " +
            "-g:lines,vars,source " +
            "-1.5 " +
            "-classpath " +
            "rt.jar" + java.io.File.pathSeparator +
            "\"" + Server.getLibDirectory() + "\"" +
            java.io.File.pathSeparator +
            "\"" + javaFile.getClassPath() + "\" " +
            "\"" + javaFile.getAbsoluteFileName() + "\"";
      ByteArrayOutputStream outputBuffer = new
            ByteArrayOutputStream();
      ByteArrayOutputStream errorBuffer = new
            ByteArrayOutputStream();
      boolean result =
            org.eclipse.jdt.internal.compiler.batch.Main.compile(cmd,
            new PrintWriter(outputBuffer),
            new PrintWriter(errorBuffer));
      parseJDTResult(errorBuffer.toString());
      if(!result)
            throw new OnlineCompilerException("Compiler failed");
      else
            javaFile.setRecompile();
```

```
        return result;
}
```

Wichtig beim Compilieren ist der Parameter „-g:lines,vars,source“. Damit werden zu den compilierten Dateien Daten hinzugefügt, die zum Debuggen des Programmes benötigt werden.

5.1.1.3.4.2 Ausführen

Das Ausführen von Java-Programmen wird von der Klasse Java*FileLaunchJobGNU* durchgeführt. Wie die Klasse zum Erstellen von Java-Programmen ist auch diese Klasse von der Klasse *Job* abgeleitet. Dass heißt, das Ausführen von Java-Programmen wird ebenfalls von einem eigenen Thread durchgeführt. Einzige Methode der Klasse ist *launchFile*, welche von dem Job-Thread ausgeführt wird:

```
protected boolean launchFile()
        throws IOException, InterruptedException,
                OnlineCompilerException
{
        String classPath = javaFile.getClassPath();
        String packagePath = javaFile.getPackagePath();
        String[] cmd;
        if(javaSecurity)
                cmd = new String[]{"java", "-Djava.security.manager",
                        "-Djava.security.policy=\"" +
                        JavaSecurityManager.secureProcess(
                        javaFile.getUser()), "-Xbootclasspath/a:" +
                        Server.getLibDirectory(), "-classpath", classPath,
                        (packagePath.equals("") ? javaFile.getClassName() :
                                packagePath + "." + javaFile.getClassName())};
        else
                cmd = new String[]{"java", "-Xbootclasspath/a:" +
                        Server.getLibDirectory(), "-classpath", classPath,
                        (packagePath.equals("") ? javaFile.getClassName() :
                        packagePath + "." + javaFile.getClassName())};
        setProcess(javaFile.getCommandLineArgs(cmd), classPath,
                JavaLanguage.processTimeout, JavaLanguage.ioCharset);
        getProcess().waitFor();
        if(isProcessTimeout(true))
                throw new TimeoutException();
        return true;
}
```

LaunchFile startet das Java-Programm und wartet, bis das Programm beendet ist. Wichtige Implementierungs-Details sind die Kommandozeilen-Parameter der Java-VM die das Java-Programm ausführen soll:

- **Djava.security.manager:**

  Aktiviert den Security-Manager. Dadurch werden Befehle wie „Datei schreiben“ gesperrt.

- **Djava.security.policy:**

  Die Policy-File des Security-Managers in der angegeben wird, welche Befehle erlaubt sind.

- **Xbootclasspath/a:**

  Verzeichnisangabe für Java-Bibliotheken

- **classpath:**

  Classpath der Anwendung. Um den korrekten Classpath zu ermitteln, wird die Package-Angabe aus der Java-Datei ausgelesen.

5.1.1.3.4.3 Debuggen

Im Gegensatz zum Compilieren und Ausführen von Programmen unterscheidet sich das Debuggen von Java-Programmen stark von dem Debuggen von C- und C++-Programmen. Java-Programme können ohne externes Programm mit einer „normalen“ Java-VM debuggt werden. Dass heißt, das zu debuggende Java-Programm wird nicht mit einem speziellem Debugger wie dem GDB ausgeführt sondern läuft in einer normalen Java-VM. Debugger-Befehle werden mit JDI *[Sun 2]*, dem Java Debug Interface, zur Java-VM gesendet.

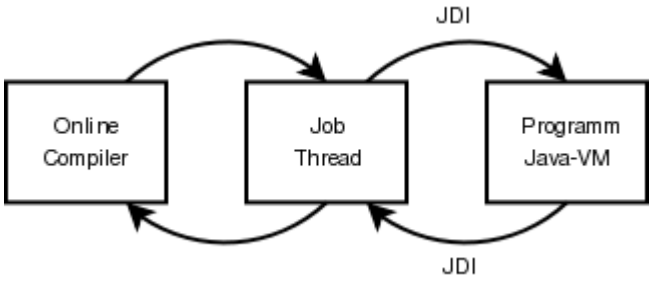


Abbildung 29: Threads und Prozesse zum Debuggen von Java-Programmen

Das JDI arbeitet Ereignis-Orientiert. Dass heißt, alle Ereignisse des zu debuggenden Java-Programms werden mittels JDI an den Job-Thread übermittelt, welcher die Ereignisse dann verarbeiten kann. Wichtige Ereignisse sind: *BreakpointEvent* falls ein Debugger-Haltepunkt erreicht wurde, *StepEvent* falls ein Step-Befehl (Step-Into, Step-Over oder Step-Out) ausgeführt wurde oder *ClassPrepareEvent,* wenn in die Java-VM eine Klasse (class-Datei) geladen wurde. Tritt ein Ereignis auf, wird das zu debuggende Java-Programm gestoppt. Mit dem JDI-Befehl *resume* wird

das Programm fortgesetzt. Der Java-Debugger-Thread (Klasse *JavaFileDebugJobSUN*) hat die gleichen Zustände, wie der C- oder C++-Debugger-Thread (Klasse *CFileDebugJobGNU*):

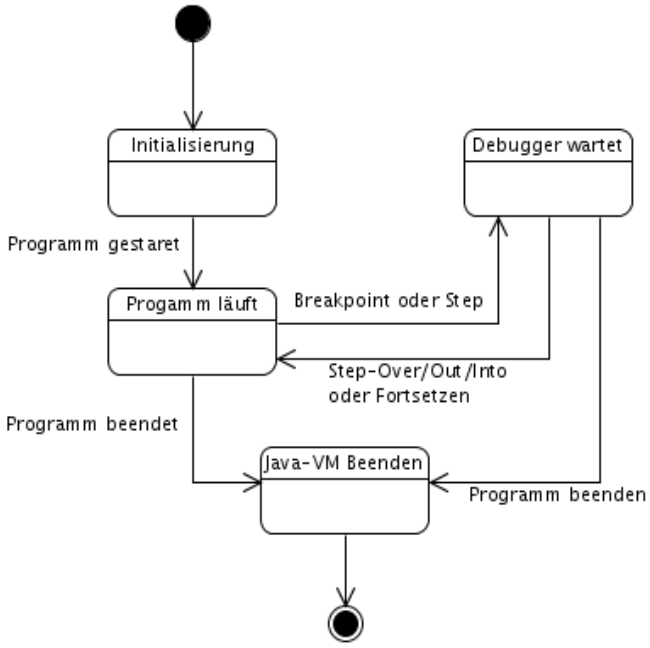


Abbildung 30: Zustände des Java-Debugger-Jobs

Nachdem das Programm gestartet wurde, wechselt der Job-Thread in den Zustand „Programm läuft“. Tritt in diesem Zustand ein Ereignis über JDI auf, wird in den Zustand „Debugger wartet“ gewechselt. Wenn das Programm beendet wurde oder zu Ende ist, wird in den Endzustand gewechselt.

Das Senden von Befehlen wie „Step-Out“ an die Java-VM des zu debuggenden Programms wird über JDI-Befehle ausgeführt.

```
public synchronized void stepInto()
     throws OnlineCompilerException
{
     if(!isDebuggerReady())
          throw new OnlineCompilerException("Debugger not
               Running");
     setDebuggerWaiting(false);
     try{
          stepRequest =
               vm.eventRequestManager().createStepRequest(
               lastThread, StepRequest.STEP_LINE,
               StepRequest.STEP_INTO);
          stepRequests.add(stepRequest);
          stepRequest.enable();
```

```
        }catch(Exception e){};
        restartWaitThread();
}
```

Um Ereignisse wie *BreakpointEvent* oder *StepEvent* von der Java-VM, die das zu debuggende Programm ausführt, zu erhalten, muss mit JDI ein so genanntes *Request*-Objekt erstellt werden. Um ein „Step-Over“-Befehl auszuführen, muss man ein *StepRequest*-Objekt erstellen und die Java-VM fortsetzen. Wenn die Java-VM dann den nächsten Java-Befehl ausgeführt hat (Step-Over), wird ein *StepEvent* erzeugt und damit zum Zustand „Debugger wartet“ gewechselt.

Ein Problem von JDI ist, das man mit JDI nicht auf Variablen der Java-VM, die das zu debuggende Programm ausführt, zugreifen kann. Mit dem GDB kann man Variablen mit dem *print*-Befehl lesen. Um Variablen mit JDI zu lesen, muss man deshalb eine weitere Komponente, das *expr*-Paket des SUN Java Debuggers (JDB) verwenden *[Sun 3]*.

```
private void updateVariable(DebugableFile.Variable variable)
{
      try {
            Value value =
                  ExpressionParser.evaluate(variable.getName(), vm,
                  this);
            variable.setValue(value.toString());
      } catch (Exception e) {
            variable.setValue();
      }
}
```

Die Methode *updateVariable* liest den Wert einer Variablen und speichert ihn in einem Variablen-Objekt. Dazu wird die Klasse *ExpressionParser* des *expr*-Paketes des JDB verwendet.

Ein weiteres Problem betrifft die Class-Dateien von Java und die Debugger-Haltepunkte. Mit JDI können nur Debugger-Haltepunkte für Java-Klassen erstellt werden. Dass heißt, man kann Debugger-Haltepunkte für eine Klasse X oder eine Klasse Y erstellen, nicht aber für Java-Dateien. Der Benutzer gibt aber Haltepunkte für einzelne Java-Dateien im Editor der Benutzerschnittstelle an. Wenn also ein Haltepunkt in einer Datei, die zwei Klassen enthält, erstellt werden soll, muss man erst herausfinden, zu welcher Klasse der Haltepunkt gehört. Das heißt, man muss den Java-Quelltext analysieren. Da dies zu aufwendig ist, wird folgendes Verfahren verwendet:

- Sobald eine Class-Datei in der Java-VM, die das zu debuggende Programm ausführt, geladen wird, tritt über JDI ein *ClassPrepareEvent* auf.
- Bei einem *ClassPrepareEvent* wird die Java-Quelltext-Datei ermittelt, in der sich die Klasse befindet. In einer Liste wird gespeichert, welche Klassen zu welcher Java-Quelltext-Datei gehören.
- Wenn ein Debugger-Haltepunkt erstellt werden soll, wird für jede Klasse, die in der Java-Datei enthalten ist, ein *BreakpointRequest* angelegt. Die Klasse, in der die Zeile des Haltepunktes

liegt, erzeugt dann bei Erreichen des Haltepunkt ein *BreakpointEvent*.

5.1.1.3.5 Lisp und C#

Die Unterstützung für Lisp und C# wurde noch nicht implementiert. Um eine spätere Implementierung aber so einfach wie möglich zu gestalten, wurden alle Klassen und Methoden, die für Lisp und C# benötigt werden, bereits erstellt.

5.1.1.3.6 AppArmor und Java-Security-Manager

Alle Programme, die von Anwendern mit dem Online-Compiler ausgeführt werden, sind potentielle Sicherheitslücken. Der Server muss deshalb vor den Anwender-Programmen „geschützt“ werden. Das heißt, um die Sicherheit des Servers zu erhöhen, müssen einige Funktionen für Anwender-Programme gesperrt werden. Dazu zählen das Aufrufen von anderen Programmen, das Schreiben in Dateien und das Lesen in bestimmten Verzeichnissen (zum Beispiel sollte im Verzeichnis `/etc` nicht gelesen werden können). Um diese Funktionen für Anwender-Programme zu sperren, werden zwei verschiedene Lösungen verwendet:

- Der **Java-Security-Manager** für Java-Programme *[Sun 4]*:

  Klasse *onlinecompiler.os.security.JavaSecurityManager*

- Das Linux-Programm **AppArmor** für C- und C++-Programme *[Novell 1]*:

  Klasse *onlinecompiler.os.security.AppArmor*

Um den Java-Security-Manager verwenden zu können, muss eine sogenannte Policy-Datei erstellt werden. In dieser Datei wird angegeben, welche Operationen für Java-Programme erlaubt sind. Die Methode *saveProfile* der Klasse *JavaSecurityManager* erzeugt die Policy-Datei für einen Anwender.

```
private static String saveProfile(User user)
      throws IOException
{
      String policyFile = user.getUserDirectory() + FileTools.sep +
            POLICY_FILE;
      FileTools.saveTextFile(policyFile,
            "grant {\n" +
            "     permission java.io.FilePermission \"" +
                  user.getRoot().getAbsoluteFileName() +
                  FileTools.getDirectorySeperator() + "*\",
                  \"read\";\n" +
            "     permission java.util.PropertyPermission \"*\",
                  \"read\";\n" +
            "     permission java.net.SocketPermission \"*\", \"\";\n"
                  +
            "};");
      return policyFile;
}
```

Die Funktionsweise von AppArmor ist fast identisch zu der Funktionsweise des Java-Security-Managers. Mit einer Policy-Datei wird festgelegt, welche Operationen für ein Programm erlaubt sind. Im Gegensatz zum Java-Security-Manager wird allerdings für jedes Programm eine eigene Policy-Datei benötigt. Die Policy-Datei eines C- oder C++-Programmes wird mit der Methode *saveProfile* der Klasse *AppArmor* erstellt.

```
private static String saveProfile(User user, String filename)
      throws IOException
{
      String profileFile = user.getUserDirectory() + FileTools.sep
            + PROFILE_FILE;
      FileTools.saveTextFile(profileFile,
            "\"" + filename + "\" {\n" +
            "#include <abstractions/base>\n" +
            "    \"" + user.getRoot().getAbsoluteFileName() +
            FileTools.sep + "**\" r,\n}");
      return profileFile;
}
```

Um AppArmor für ein Programm zu aktivieren, muss man den Befehl „apparmor_parser --replace Policy-Datei“ aufrufen. Dazu dient die Methode *secureProcess* der Klasse *AppArmor*. Die Methode erzeugt zunächst die Policy-Datei des C- oder C++-Programmes und ruft anschließend den „apparmor_parser“ Befehl mit der Policy-Datei auf.

```
public synchronized static void secureProcess(User user, String
      filename)
      throws CreateSandboxException
{
      Server.debugMessage("Create AppArmor Sandbox for " +
            filename);
      try{
            Process p = Runtime.getRuntime().exec(new
                  String[]{"apparmor_parser", "--replace",
            saveProfile(user, filename)});
            p.waitFor();
            p = null;
      }catch(Exception e){
            Server.errorMessage(e);
            throw new CreateSandboxException();
      }
}
```

Im Gegensatz zum Java-Security-Manager muss bei AppArmor der Schutz für ein bestimmtes Programm explizit wieder deaktiviert werden. Dazu dient die Methode *unsecureProcess* der Klasse *AppArmor*. Diese Methode muss beim Löschen eines C- oder C++-Programmes aufgerufen werden.

```
public synchronized static void unsecureProcess(User user, String
      filename)
{
      Server.debugMessage("Delete AppArmor Sandbox for " +
            filename);
      try{
```

```
            Process p = Runtime.getRuntime().exec(new
                  String[]{"apparmor_parser", "--remove",
            saveProfile(user, filename)});
            p.waitFor();
      }catch(Exception e){
            Server.errorMessage(e);
      }
}
```

Die Methode erzeugt zunächst die Policy-Datei des C- oder C++-Programmes und ruft anschließend den „apparmor_parser“ Befehl mit der Policy-Datei und dem Parameter „--remove“ (zum entfernen des AppArmor-Schutzes) auf.

5.1.1.3.7 User und Client-Sessions

Die Klasse *User* und die von ihr abgeleiteten Klassen dienen zum Speichern aller Informationen eines Users des Online-Compilers. Im Gegensatz zum Java-Online-Compiler, wo für jeden Client ein eigener User angelegt wurde, müssen beim Online-Compiler die User-Objekte und die Client-Verbindungen (Sessions) getrennt werden:

- Eine Session kann nacheinander verschiedenen Usern zugeordnet werden da sich der Benutzer an- und abmelden kann ohne die Verbindung zu trennen (Schließen des Webbrowsers).
- Mehrere Client-Verbindungen (Sessions) können dem selben User zugewiesen werden da sich ein Anwender an verschiedenen Webbrowsern mit dem gleichen Benutzernamen und Passwort anmelden kann.

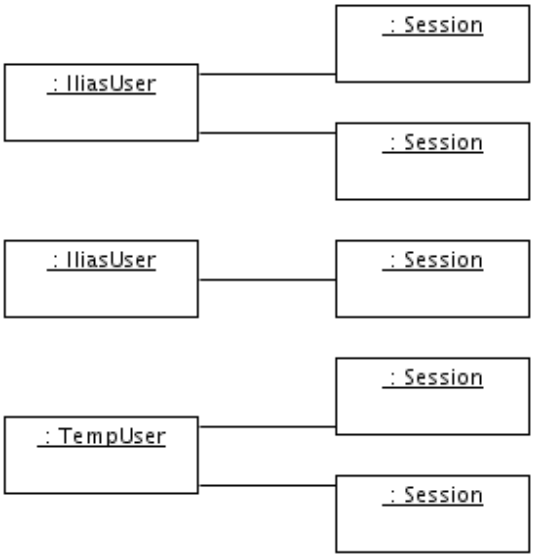


Abbildung 31: Objektdiagramm Sessions und Users

Mehrere Sessions können als auf den gleichen User zugreifen. Mit der Methode *registerSession* der Klasse *User* wird eine *Session* einem *User* zugeordnet. Das Zuordnen von einer *Session* zu einem *User*-Objekt erfolgt erst nach dem der Benutzer die Anmeldedaten in das entsprechende Formular der Benutzerschnittstelle eingegeben hat.

```
public synchronized void registerSession(Session session)
{
      if(session != null)
            addFileObjectListener(session);
}
```

Wenn Sessions sich an einen *User* anmelden können, muss es auch eine Möglichkeit geben, *Sessions* von einem *User* wieder abzumelden. Dazu wird die Methode *unregisterSession* der Klasse *User* verwendet:

```
public synchronized void unregisterSession(Session session)
{
     removeFileObjectListener(session);
     if(canDestroy())
          delete();
}
```

Mit dem Befehl *delete* wird ein *User*-Objekt gelöscht. Dass heißt, die Ressourcen des Users (zum Beispiel die Dateien) werden gelöscht. Dies darf natürlich nur dann passieren, wenn keine Session mehr auf den User zugreift. Dies wird mit der Methode *canDestroy* überprüft. Eine Ausnahme bildet der *AdminUser*. Ein Administrator-User dient zum Speichern der Beispiele. Da alle Benutzer des Online-Compilers auf die Beispiele zugreifen sollen, dürfen *AdminUser*-Objekte nicht gelöscht werden. Aus diesem Grund ist die Methode *canDestroy* in der Klasse *AdminUser* überschrieben:

```
protected synchronized boolean canDestroy()
{
     return false;
}
```

Die Freigabe der Ressourcen eines *User*-Objektes erfolgt durch die Methode *delete* der Klasse *User*:

```
public synchronized void delete()
{
     if(isDeleted())
          return;
     deleted = true;
     removeUser(this);
     if(storeFiles)
          saveFiles();
     root.delete();
     FileTools.deleteDirectory(root.getAbsoluteFileName());
}
```

Mit dem Befehl *delete* des Objektes *root* werden alle Quelltextdateien und Verzeichnisse, die vom User erstellt wurden, gelöscht. Vor dem Löschen werden allerdings (außer bei Gästen bzw. der Klasse *TempUser*) die Dateien und deren Konfiguration in einer Datei gespeichert damit sie beim nächsten Anmelden des Users wieder verwendet werden können. Die Daten aller User werden in einer Datei mit dem Namen `files.xml` gespeichert. Zum Speichern dieser Dateien werden die gleichen Funktionen verwendet, die auch zum Download von XML-Dateien verwendet werden.

5.1.1.3.7.1 Ereignisse

Ereignisse sind zum Beispiel das Erstellen oder Löschen einer Datei. Tritt in einem *User*-Objekt ein Ereignis auf, müssen alle Sessions, die an das *User*-Objekt angemeldet sind, benachrichtigt werden. Dazu wird zum Beispiel die Methode *fireObjectCreated* der Klasse *User* verwendet. Diese Methode wird aufgerufen, wenn eine Datei oder ein Verzeichnis erstellt wurde:

```
public void fireFileObjectCreated(FileObject fo)
{
     for(int i = 0; i < listener.size(); i++)
          if(listener.get(i) != null)
               listener.get(i).fileObjectCreated(fo);
}
```

Die Methode ruft in jeder angemeldeten Session die Methode *fileObjectCreated* auf. Diese Methode speichert das Ereignis in einer Warteschlange (Klasse *EventQueue*). Bei Client-Anfragen werden diese Ereignisse aus der Warteschlange zum Client übertragen. Auf diese Weise erhalten alle Clients die bzw. deren Sessions an ein bestimmtes *User*-Objekt angemeldet sind, alle Ereignisse des Users.

```
public void fileObjectCreated(FileObject fo)
{
     if(fo instanceof RootFolder)
          return;
     fileEvents.addEvent(SessionEvent.FILE_CREATE, fo);
}
```

5.1.1.3.7.2 Session Synchronisation

Der Tomcat-Webserver führt alle Client-Anfragen in einem eigenen Thread aus. Dass heißt, dass mehrere Anfragen von mehreren Clients, die auf das gleiche *User*-Objekt zugreifen, parallel ausgeführt werden können. Mehrere Threads können also gleichzeitig auf die gleichen Daten zugreifen wodurch die Gefahr von inkonsistenten Daten (fehlerhafte Daten da mehrere Threads gleichzeitig die Daten bearbeitet haben) besteht. Es ist deshalb eine Synchronisation der Threads notwendig.

```
private Boolean sessionLock = new Boolean(false);

/** True, wenn Session das User-Objekt f&uuml;r andere Sessions
gesperrt hat. */
private boolean sessionLocked = false;

/**
 * Sperrt den User f&uuml;r andere Sessions.
 * Wenn User bereits gesperrt ist, wird gewartet,
 * bis der User wieder freigegeben wird.
 */
public void lockForSession()
{
     synchronized(sessionLock)
     {
          while(sessionLocked)
               try{
```

```
                    sessionLock.wait();
                }catch(Exception e){};
            sessionLocked = true;
        }
}

/**
 * Gibt den User f&uuml;r andere
 * Sessions frei.
 */
public void unlockForSession()
{
      synchronized(sessionLock)
       {
            sessionLocked = false;
              try{
                 sessionLock.notify();
            }catch(Exception e){};
      }
}
```

Vor jedem Zugriff einer Session auf *User*-Daten muss die Methode *lockForSession* der Klasse *User* aufgerufen werden. Dadurch wird der User für andere Sessions gesperrt. Dass heißt, greift eine andere Session auf den User zu, muss sie warten bis der User wieder freigegeben wurde. Das Freigeben eines Users, nachdem alle Operationen abgeschlossen sind, erfolgt durch die Methode *unlockForSession* der Klasse *User*.

Neben den Sessions greifen aber auch noch die Jobs auf User-Daten zu. Das heißt, die Job-Threads müssen ebenfalls mit den Sessions synchronisiert werden. Dies kann aber nicht auf gleiche Weise mit den beiden obigen Methoden erfolgen. Wenn ein Job-Thread ein *User*-Objekt sperrt, müssten alle Sessions warten, bis der Job beendet ist. Das heißt, wenn ein Job zum Beispiel 10 Minuten dauern würde, müssten alle Sessions (und damit auch die Clients), die auf das gleiche *User*-Objekt zugreifen, mindestens 10 Minuten warten.

Wenn ein Job für einen User aktiv ist (zum Beispiel ein *CompileJob* um eine Datei des Users zu compilieren), wird der Job-Thread in dem Attribut *job* des *User*-Objektes gespeichert. Wenn eine Session eine Client-Anfrage bearbeiten soll, wird zunächst die Methode *checkUserJobRunning* aufgerufen.

```
public void checkUserJobRunning()
      throws UserJobIsRunningException
{
      if(job != null){
            if(!job.isStopped())
                  throw new UserJobIsRunningException();
            else
```

```
                removeJob(job);
        }
}
```

Diese Methode erzeugt genau dann eine Ausnahme, wenn ein Job-Thread gerade aktiv ist. Die Abarbeitung einer Client-Anfrage durch ein *Session*-Objekt wird also genau dann mit einer Fehlermeldung abgebrochen, wenn ein Job gerade aktiv ist. Wenn ein Job aktiv ist, blockieren die Sessions dadurch aber nicht und beantworten Client-Anfragen weiterhin (allerdings mit einer Fehlermeldung bis der Job zu Ende ist).

5.1.1.3.8 Kommunikation Client-Servlet

Der Tomcat-Webserver ruft bei Client-Anfragen die Methode *doPost* der Klasse *OCServlet* auf. Jede Client-Anfrage wird in einem eigenen Thread ausgeführt. Es können also mehrere Client-Anfragen gleichzeitig behandelt werden.

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
     throws ServletException, java.io.IOException
{
     HttpSession httpSession = request.getSession(true);

     // Sprache prüfen
     String language = ServletTools.getStringCookie(request,
          "OC_LANG", FileTranslator.DEFAULT_LANGUAGE);
     language = ServletTools.getStringParameter("setlang",
          request, language).toLowerCase();
     ServletTools.setCookie(response, "OC_LANG", language, true);

     if(isDocumentRequest(request))
          answerDocumentRequest(request, httpSession, response,
               language);
     else {
          try{
               Session clientSession = Session.getSession(request,
                    httpSession);
               ...
               clientSession.answerSessionRequest(request,
                    response);
          }catch(MaxSessionCountReachedException e1){
               try{
                    Session.answerRequestWithError("Max Session
                         Count Reached", response);
               }catch(Exception e2){
                    Server.errorMessage(e2);
               }
          }catch(Exception e){
               Server.errorMessage(e);
          }
     }
}
```

Nachdem am Anfang der Methode die Sprache der Benutzerschnittstelle aus der Client-Anfrage oder aus Cookies des Webbrowsers ausgelesen wurde, wird mit der if-Anweisung geprüft, ob eine XML-Anfrage (Ajax) gestellt wurde oder ob der Client eine HTML- oder JavaScript-Datei angefordert hat. Falls eine Datei angefordert wurde, wird der Dateiinhalt in der gewünschten Sprache mit der Methode *answerDocumentRequest* zum Client gesendet. XML-Anfragen werden an

das Session-Objekt, welches der Verbindung zugewiesen wurde, weitergeleitet.

5.1.1.3.8.1 XML-Antworten des Servlets

XML-Anfragen von Clients werden von der Methode *answerSessionRequest* der Klasse beantwortet. Die Methode *answerSessionRequest* führt folgende Operationen aus:

- Parsen der XML-Anfrage, bei einem Format-Fehler in der XML-Anfrage wird eine Fehlermeldung zum Client gesendet.
- Erstellen eines neuen XML-Dokumentes für die Antwort und Durchlaufen der XML-Anfrage mit einer Schleife.
- Jeder Befehl (siehe nächster Abschnitt „Kommunikationsprotokoll“) wird in einer eigenen Methode abgearbeitet. Das Ergebnis des Befehls wird mit einem neuen XML-Knoten zur Antwort hinzugefügt.
- Die Antwort (XML-Dokument) wird zum Client gesendet.

Jeder Befehl des Kommunikationsprotokolls wird in einer eigenen Methode abgearbeitet. Die Methode *commandDebugFile* dient zum Beispiel zur Bearbeitung des Befehls „debug-file“, mit dem der Debugger für eine Programm-Datei gestartet werden kann.

```
1:   private void commandDebugFile(Document responseXML, Element
2:         commandsNode, Element command)
3:         throws OnlineCompilerException
4:   {
5:         lockUser();
6:         user.checkUserJobRunning(); // kein Job darf laufen
7:         String ID = command.getAttribute("id");
8:         FileObject fo = getFileObjectByID(ID);
9:         if(fo == null) throw new InvalidIDException();
10:        user.addJob(job = fo.debug());
11:        job.addListener(this);
12:        job.start();
13:        Element result = createXMLCommandResult(CMD_DEBUGFILE,
14:              responseXML);
15:        result.setAttribute("id", ID);
16:        commandsNode.appendChild(result);
17:  }
```

Am Anfang der Methode wird zunächst das *User*-Objekt für andere Sessions und Client-Anfragen gesperrt (Zeile 5). Danach wird geprüft, ob bereits ein Job-Thread aktiv ist (Zeile 6). Falls ja, wird eine Ausnahme geworfen und die Abarbeitung des Befehls abgebrochen. In den Zeilen 7 bis 9 wird die Datei gesucht, die debuggt werden soll. In Zeile 10 wird der Job erstellt und in Zeile 12 der Thread gestartet. In den Zeilen 13 bis 16 wird das Ergebnis des Befehls zur XML-Antwort hinzugefügt.

5.1.1.3.8.2 Kommunikationsprotokoll

Wie bereits im Entwurf festgelegt, werden Nachrichten zwischen Client und Servlet im XML-Format übertragen. Eine Nachricht vom Client zum Servlet ist immer wie folgt aufgebaut:

<oc-request>

<befehl1 parameter1="..." parameter2="..." />

<befehl2 parameter1="..." parameter2="..." />

<befehl3 parameter1="..." parameter2="..." />

…

</oc-request>

Eine Antwort des Servlets an den Client hat immer folgendes Format:

<oc-response>

<befehl1>...</befehl1>

<befehl2>...</befehl2>

<befehl3>...</befehl3>

…

</oc-response>

Wenn ein Befehl aufgrund eines Fehlers nicht ausgeführt werden konnte, wird die Ausführung der anderen Befehle abgebrochen. Der Befehl, der nicht ausgeführt werden konnte, wird durch das Attribut „success=false" gekennzeichnet. Die Fehlerursache wird in dem XML-Knoten, der für den Befehl verwendet wird, eingefügt:

Eine Antwort bei einem Fehler im 2. Befehl ist folgendermaßen aufgebaut:

<oc-response>

<befehl1>...</befehl1>

<befehl2 success=false>Fehlermeldung</befehl2>

</oc-response>

5.1.1.3.8.3 Befehlsübersicht

Im Vergleich zum Java-Online-Compiler sind beim Online-Compiler 17 Befehle mehr vorhanden. Diese werden hauptsächlich zur Steuerung des Debuggers benötigt. Bei vielen Befehlen müssen Dateien oder Verzeichnisse als Parameter angegeben werden. Dafür existieren 3 verschiedene Möglichkeiten:

- Angabe der **ID** der Datei oder des Verzeichnisses. Jede Datei und jedes Verzeichnis hat eine eindeutige Nummer, die sogenannte ID
- Angabe des **Pfades** der Datei oder des Verzeichnisses (z.B. `/a/b/c/d.txt`)
- Angabe des Kurz-Namens (**sname**) der Datei oder des Verzeichnisses. Jede Datei kann einen sname haben. So kann man zum Beispiel für jedes Programm-Beispiel einen eindeutigen Namen (sname) vergeben (z.B. „da_beispiel_1“, „da_beispiel_2“ usw.). Zum Aufruf eines Programm-Beispiels kann man dann einfach „da_beispiel_1“ statt „`/Beispiele/Datenstrukturen + Algorithmen/Beispiel 1"` angeben.

In den folgenden Tabellen werden alle vorhandenen Befehle beschrieben. Parameter die *kursiv* geschrieben werden, müssen innerhalb des Befehls-Tags angegeben werden:

<befehl1 …>*Parameter*</befehl1>

Befehle zum An- und Abmelden und zur Verbindungs-Verwaltung:

Befehl	***Beschreibung***	***Parameter***	***Rückgabewert***
destroy-session	Trennt eine Verbindung und gibt alle Ressourcen der Verbindung sofort frei (wird nur zum Test benötigt)	keine	kein
open-session	Beginnt eine Client-Session	keine	kein
close-session-at-end	Beendet eine Client Session auch wenn ein Fehler aufgetreten ist	keine	kein
close-session	Beendet eine Client-Session	keine	kein
login	Anmeldung als Benutzer	name: Name des Benutzers (optional) password: Passwort des Benutzers (optional) guest: true, falls man sich als Gast anmelden möchte (optional) autologin: true, falls zum Login die zuletzt eingegebenen Werte verwendet werden sollen (optional)	Name des Benutzers

Befehl	*Beschreibung*	*Parameter*	*Rückgabewert*
logout	Abmelden als Benutzer	keine	kein
upload-result	Liefert das Ergebnis des letzten Datei-Uploads zurück	keine	Fehlermeldung falls Upload fehlgeschlagen

Befehle zur Verwaltung von Dateien:

Befehl	*Beschreibung*	*Parameter*	*Rückgabewert*
create	Erstellt eine neue Datei oder ein neues Verzeichnis	name: Name der neuen Datei parent: ID des übergeordneten Verzeichnisses type: Typ der Datei (java, c++, c#, header, c, lisp, folder oder text) exe: true, falls neue Datei ein ausführbares Programm enthalten soll (optional)	ID der Erstellten Datei
delete-file	Löscht eine Datei oder ein Verzeichnis	id: Datei-ID	kein
rename	Benennt eine Datei oder ein Verzeichnis um	newname: neuer Name id: Datei-ID	neuer Datei oder Verzeichnisname
request-file-changes	Liefert alle Datei-Ereignisse	ack: Nummer des Ereignisses, welches zuletzt empfangen wurde (Fenster) all: true, falls alle vorhandenen Dateien übertragen werden sollen (optional), sinnvoll beim Start	Datei-Ereignisse (siehe unten)

Befehle zum Bearbeiten von Dateiinhalten:

Befehl	***Beschreibung***	***Parameter***	***Rückgabewert***
open-file	Öffnet eine Datei	id: Datei-ID parent: ID des Verzeichnisses, in dem nach der Datei gesucht werden soll (optional)	Dateiinhalt, Dateitargets, Haltepunkte und Kommandozeilen-Argumente
add-variable	Erstellt eine neue Debugger-Variable	id: Datei-ID name: Variablen-Name	Wert der Variablen
delete-variable	Löscht eine Debugger-Variable	id: Datei-ID name: Variablen-Name	kein
save-file-targets	Speichert die Targets einer Datei	id: Datei-ID *Targets* (siehe unten)	kein
save-file-args	Speichert die Kommandozeilenargumente einer Datei	id: Datei-ID *Kommandozeilenargumente*	kein
save-file	Speichert den Inhalt und die Haltepunkte einer Datei	id: Beispiel-ID *Dateiinhalt* (siehe unten) *Haltepunkte* (siehe unten)	kein

Befehle für Beispiele:

Befehl	***Beschreibung***	***Parameter***	***Rückgabewert***
request-examples	Liefert alle Beispiele die es gibt zurück	keine	Liste aller Beispiele
open-example	Lädt ein Beispiel	id: Beispiel-ID	ID des geladenen Beispiels

Befehle für Jobs:

Befehl	***Beschreibung***	***Parameter***	***Rückgabewert***
build-file	Startet einen neuen Build-Job	id: Datei-ID	kein
launch-file	Startet einen neuen Launch-Job	id: Datei-ID	kein
debug-file	Startet einen neuen Debug-Job	id: Datei-ID	kein
stop-job	Stoppt den aktuellen Job	keiner	kein
input	Eingabe für laufenden Prozess (bei Debug- oder Launch-Job)	*Eingabe*	kein
debug-step-out	Step-Out Befehl bei Debug-Job	keiner	kein
debug-step-over	Step-Over Befehl bei Debug-Job	keiner	kein
debug-step-into	Step-Into Befehl bei Debug-Job	keiner	kein
debug-resume	Step-Resume Befehl bei Debug-Job	keiner	kein
request-job-changes	Liefert alle Job-Ereignisse	ack: Nummer des Ereignisses, welches zuletzt empfangen wurde (Fenster)	Job-Ereignisse (siehe unten)

Der Befehl *request-file-changes* liefert alle Änderungen, die seit dem letzten Aufruf des Befehls an den Dateien vorgenommen wurden. Im Entwurf wurde dazu bereits drei Ereignisse festgelegt:

- eine Datei oder ein Verzeichnis wurde erstellt:

  für solche Ereignisse wird zum Client **<file-create …>** gesendet. In diesem XML-Knoten sind alle Eigenschaften der Datei bzw. des Verzeichnisses enthalten

- eine Datei oder ein Verzeichnis wurde verändert:

  für solche Ereignisse wird zum Client **<file-change …>** gesendet. In diesem XML-Knoten sind alle Eigenschaften der Datei bzw. des Verzeichnisses enthalten

- eine Datei oder ein Verzeichnis wurde gelöscht:

  für solche Ereignisse wird zum Client **<file-delete …>** gesendet.

Der Befehl *request-job-changes* liefert alle Änderungen, die seit dem letzten Aufruf des Befehls an den laufenden Job vorgenommen wurden. Im Entwurf wurden dazu bereits fünf Ereignisse festgelegt:

- ein Job wurde gestartet:

  für solche Ereignisse wird zum Client **<job-started …>** gesendet.

- ein Job wurde beendet:

  für solche Ereignisse wird zum Client **<job-finished …>** gesendet.

- der Debugger wurde angehalten:

  für solche Ereignisse wird zum Client **<job-debugger-waiting …>** gesendet.

- eine Fehlermeldung oder eine Warnung des aktuellen Jobs:

  für solche Ereignisse wird zum Client **<job-result …>** gesendet.

- Ausgabe vom aktuell laufendem Prozess ist vorhanden:

  für solche Ereignisse wird zum Client **<job-output …>** gesendet.

Beispiel für den Befehl request-job-changes:

Wenn ein Job (z.B. mit dem Befehl *build-job*) gestartet wurde und ohne Fehlermeldung abgeschlossen wird, erhält man durch den Aufruf von *request-job-changes* folgende Antwort vom Servlet:

```
<oc-response>
    <request-job-changes>
        <job-started />
        <job-finished />
    </request-job-changes>
</oc-response>
```

5.1.1.3.9 Log-Datei

Bei besonderen Ereignissen wie Ausnahmen (Exceptions) wird ein Meldung in der Log-Datei des Tomcat-Servers gespeichert. Die Log-Datei (*catalina.out*) befindet sich im Unterverzeichnis *logs* des Tomcat-Servers. Es gibt drei Arten von Meldungen:

- Fehlermeldungen: bei einer Exception
- Statusmeldungen: bei sonstigen wichtigen Ereignissen
- Debug-Meldungen: wird nur zur Fehlersuche verwendet

Die verschiedenen Meldungen können in der Konfigurationsdatei *server.cfg* des Online-Compiler-Servlets aktiviert oder deaktiviert werden. Zum Aufzeichnen von Meldungen dienen die Methoden *logMessage*, *errorMessage* und *debugMessage* der Klasse *Server*.

Methode *errorMessage* der Klasse *Server*:

```
public static void errorMessage(String message)
{
      if(logError){
            System.out.println("OC ERROR:");
            System.out.println(message);
            System.out.println();
      }
}
```

5.1.2 Das Webinterface

Ziel dieses Kapitels ist die Beschreibung von Implementierungsdetails der Benutzerschnittstelle (Webinterface) des Online-Compilers. Die Benutzerschnittstelle wurde komplett mit Hilfe von JavaScript und HTML implementiert.

Neben diesem Kapitel steht auch noch eine ausführliche **Inline Dokumentation** in den JavaScript-Quelldateien zur Verfügung (Verzeichnis `source/OnlineCompiler/oc` auf der CD).

5.1.2.1 Datei-Übersicht

Folgende Dateien aus dem Verzeichnis `source/OnlineCompiler/oc` gehören zu der Benutzerschnittstelle (Webinterface) des Online-Compilers:

Datei- oder Verzeichnisname	*Beschreibung*
ace.js	Stellt die Verbindung zum Servlet her
console.js	Grafische Komponente zum Anzeigen von Ein- und Ausgabe von Programmen
debugger.js	Grafische Komponente zum Anzeigen von Informationen des Debuggers
docsearchdialog.js	Dialog zum Suchen in Dokumentationen von Programmiersprachen
fileargsdialog.js	Dialog zum Bearbeiten von Kommandozeilenargumenten
filetargetsdialog.js	Dialog zum Bearbeiten von Compiler- und Linker-Targets
filetree.js	Grafische Komponente zum Anzeigen von Dateien und Verzeichnissen (Dateibaum)
gui.js	Hauptdatei: enthält Funktionen zum Zugriff auf das Servlet und das eigentliche Programm
guitools.js	Sammlung von Funktionen die zum Erstellen einer Grafischen Benutzeroberfläche dienen
includeos.js	Funktionen zur Integration des Online-Compilers in andere Web-Seiten (z.B. in Ilias)
logindialog.js	Dialog zum Eingeben von Benutzername und Passwort
maintab.js	Grafische Komponente zum Anzeigen von verschiedenen Tabs
menu.js	Grafische Komponente zum Anzeigen eines Menüs
oc.js	Modellklassen die Daten des Online-Compilers speichern (MVC)
problems.js	Grafische Komponente zum Anzeigen von Fehlermeldungen und Warnungen
sourcefileeditor.js	Grafische Komponente: Editor für Quelltextdateien
splitter.js	Grafische Komponente zum ändern der Größe von anderen Komponenten
uploaddialog.js	Dialog zum Auswählen einer Datei zum Upload
index.html	eigentliche Web-Seite
indexklein.html	Web-Seite der Ilias-Version des Online-Compilers
unsupportedbrowser.html	Fehlerseite falls Browser nicht unterstützt wird
upload.html	Formular zum Hochladen einer Datei zum Servlet (wird für den IE benötigt)

5.1.2.2 Implementierung der Benutzerschnittstelle

Die Benutzerschnittstelle soll als Ajax-Anwendung implementiert werden. Dies bedeutet, dass der Client (bzw. die Benutzerschnittstelle) vom Server Daten im XML-Format bekommt und diese verarbeitet. Der Client muss also eine eigene Funktionalität besitzen beziehungsweise ein richtiges Programm sein. Herkömmliche Webanwendungen sind als statische Webseiten implementiert. Das heißt, die komplette Funktionalität befindet sich auf einem Webserver der die Webseiten generiert die dann auf dem Client angezeigt werden. Bei einer Ajax-Anwendung wird ein Teil der „Arbeit“ vom Client selbst ausgeführt.

Mithilfe der Programmiersprache JavaScript kann man in Webseiten (HTML-Dokumente) Programme integrieren, die dann auf den Clients, die die Webseiten anzeigen, ausgeführt werden.

5.1.2.2.1 Dynamisches HTML

Wie bereits mehrfach erwähnt wurde, muss der Client XML-Antworten des Servlets verarbeiten und ggf. die Benutzerschnittstelle aktualisieren. So muss zum Beispiel eine Compiler-Fehlermeldung, die der Client vom Servlet empfangen hat, in die entsprechende Komponente der Benutzerschnittstelle angezeigt werden. Da die Benutzerschnittstelle ein HTML-Dokument ist, muss dazu der Inhalt des Dokuments verändert werden. Dazu wird JavaScript benötigt.

JavaScript bietet grundsätzlich zwei Möglichkeiten, ein HTML-Dokument zu verändern:

- Ersetzen von HTML-Code mit der JavaScript-Funktion *innerHTML*
- Verändern von HTML-Code durch Manipulation des sogenannten DOM (Document Object Modell), einer Programmierschnittstelle zum Zugriff auf ein HTML-Dokument

Bei der ersten Methode mit der JavaScript-Funktion *innerHTML* kann man Teile eines HTML-Dokumentes durch anderen HTML-Code ersetzen. Ein Vorteil dieser Methode liegt in dessen Einfachheit. Um den Inhalt einer HTML-Seite durch einen anderen Inhalt zu ersetzen, muss man nur den folgenden JavaScript-Befehl ausführen:

```
document.body.innerHTML = "<h1>Hallo Welt</h1>";
```

Durch den Befehl wird im Web-Browser der Text „Hallo Welt“ angezeigt. Ein weiterer Vorteil der Methode ist die Geschwindigkeit. Vor allem der Internet-Explorer hat mit der zweiten Methode Probleme so das teilweise, zumindest bei älteren Versionen des Internet-Explorers, die erste Methode mit *innerHTML* um den Faktor 100 schneller ausgeführt wird als die zweite Methode *[Koch 2006]*.

innerHTML hat aber auch Nachteile: so kann man zwar einfach HTML-Code ersetzten, ein Zugriff auf einzelne HTML-Elemente und dessen Attribute ist aber nicht möglich. So kann man zum

Beispiel die Farbe eines HTML-Elementes nur dadurch ändern, dass man den kompletten HTML-Code ersetzt.

DOM (Document Object Modell) ist eine Datenstruktur (ein Baum), in der alle Elemente eines HTML-Dokumentes als JavaScript-Objekte gespeichert sind. Auf die Objekte und deren Attribute kann man mit JavaScript zugreifen. Außerdem kann man die Datenstruktur verändern und auf diese Weise HTML-Elemente hinzufügen, löschen oder verschieben. Der große Vorteil dieser Methode liegt darin, dass man direkten Zugriff auf alle Eigenschaften und Elemente eines HTML-Dokumentes hat und das alle HTML-Elemente auch JavaScript-Objekte sind. Man kann dadurch JavaScript-Programme entwickeln, die Programmen in objektorientierten-Programmiersprachen wie zum Beispiel Java (mit Swing) ähneln.

Das obige *innerHTML*-Beispiel mit Hilfe von DOM:

```
h1 = document.createElement("h1");
h1.appendData(document.createTextNode("Hallo Welt"));
document.body.appendChild(h1);
```

h1 ist das DOM-Objekt des HTML-Elementes. Mit Methoden und Attributen kann man auf die Eigenschaften des h1-Elementes zugreifen. Um die Farbe zu ändern, kann man einfach den folgenden JavaScript-Befehl aufrufen:

```
h1.style.color = "red";
```

Um die Farbe mit der innerHTML-Methode zu ändern, bleibt einem nichts anderes übrig, als den kompletten HTML-Code zu ersetzen:

```
document.body.innerHTML =
     "<h1 style=\“color: red;\“>Hallo Welt</h1>";
```

Wie man leicht sehen kann, ist das Verändern von Attributen mit *innerHTML* weit weniger komfortabel.
Bei der Implementierung habe ich mich für die DOM-Methode zur Manipulation der Benutzerschnittstelle entschieden. Die Vorteile dieser Methode im einzelnen:

- einfache Integration in JavaScript
- objektorientiert
- fehlerhafter HTML-Code wird schneller erkannt da durch falschen HTML-Code (DOM) auch immer ein JavaScript-Syntax-Fehler verursacht wird

5.1.2.2.2 HTML-Code „einpflanzen“

Oben wurde bereits folgendes Beispiel zur Änderung von HTML-Dokumenten angegeben:

```
h1 = document.createElement("h1");
h1.appendData(document.createTextNode("Hallo Welt"));
document.body.appendChild(h1);
```

In diesem Beispiel wird nur ein HTML-Element (ein h1-Element) erstellt. Die Benutzerschnittstelle ist aber eine größere Anwendung mit einem komplexen HTML-Code. So besteht zum Beispiel eine einzige Compiler-Fehlermeldung aus bis zu 30 einzelnen HTML-Elementen. Der Umfang der JavaScript-Befehle zum Erzeugen einer Compiler-Fehlermeldung wäre also ungefähr das 30-fache des obigen Beispiels. Um das Erzeugen solch komplexer HTML-Strukturen zu vereinfachen, gibt es eine spezielle JavaScript-Funktion: *graft* (engl. einpflanzen).

Beispiel Funktion *graft*:

```
graft(this.editorTab, null,
        ['b',
                ['center',
                        ['img',
                                {border:0, src:"img/load_ani.gif"}
                        ]
                ]
        ]
);
```

Den gleichen HTML-Code kann man natürlich auch ohne die Funktion *graft* erstellen. Der JavaScript-Quelltext ist aber wesentlich unübersichtlicher als der obige Aufruf von *graft*:

```
b = document.createElement("b");
this.editorTab.appendChild(b);
center = document.createElement("center");
b.appendChild(center);
img = document.createElement("img");
center.appendChild(img);
img.border = 0;
img.src= "img/load_ani.gif";
```

Die Funktion *graft* erleichtert das Erstellen von HTML-Code:

- mehrere HTML-Elemente können mit einem Befehl erstellt werden
- HTML-Elemente können verschachtelt werden
- der JavaScript-Quelltext wird übersichtlicher

5.1.2.2.3 Ereignisbehandlung

In den beiden vorherigen Abschnitten wurde auf das Erstellen und Verändern der Benutzerschnittstelle eingegangen. Ein weiterer wichtiger Aspekt von Benutzerschnittstellen ist das Behandeln von Benutzereingaben wie Mausklicks und Tastatureingaben, sogenannte Ereignisse. In eine Web-Browser können bei jedem HTML-Element Ereignisse auftreten. Wenn man zum Beispiel auf einen Link in einer HTML-Seite klickt, wird ein Ereignis ausgelöst. Das gleiche passiert zum Beispiel, wenn man die Maus über ein HTML-Element bewegt. Um ein Ereignis zu verarbeiten beziehungsweise um auf ein Ereignis zu reagieren, muss man einen sogenannten Ereignis-Handler an das HTML-Element und das gewünschte Ereignis anmelden. Ein Ereignis-Handler ist eine JavaScript-Funktion, die aufgerufen wird, sobald das Ereignis eintritt. In JavaScript gibt es zwei Möglichkeiten, Ereignis-Handler an HTML-Elemente und Ereignisse anzumelden:

- Die JavaScript-Funktion *attachEvent* (im Internet-Explorer) beziehungsweise *addEventListener* (alle anderen Web-Browser) zum Hinzufügen von Ereignis-Handlern zu HTML-Elementen
- Verwendung von Ereignis-Attributen der HTML-Elemente zum Setzten der Ereignis-Handler.

Beide Methoden sind theoretisch identisch. Leider verhalten sich die beiden Methoden nicht bei jedem HTML-Element so wie man es erwartet so das man je nach HTML-Element entweder die eine oder nur die andere Methode verwenden kann. So muss zum Beispiel für Ereignisse von HTML-IFrame-Elementen im Internet-Explorer der Ereignis-Handler mit dem Ereignis-Attribut hinzugefügt werden. Im Firefox-Browser hingegen muss man für HTML-IFrame-Elemente die *addEventListener*-Funktion verwenden.

Ein weiteres Problem ist, dass man für jedes Ereignis nur einen Ereignis-Handler verwenden kann. Dass heißt, wenn man bei einem Ereignis eine weitere Funktion aufrufen möchte, kann man diese nicht als Ereignis-Handler an das HTML-Element anmelden.

Ein weiteres Problem ist, das man bei Ereignissen in JavaScript keine objektorientierte-Programmierung verwenden kann. Ereignis-Handler sind in JavaScript immer einfache Funktionen (also keine Objekt-Methoden).

Um diese und weitere Probleme zu lösen, wurde die JavaScript-Klasse *OCGUIEvent* entwickelt welche die komplette Ereignis-Behandlung der Online-Compiler-Benutzerschnittstelle durchführt. Dadurch, dass die Ereignis-Behandlung von einer zentralen Klasse durchgeführt wird, erhält man einen großen Vorteil: Änderungen betreffen nur einen kleinen Teil des gesamten Programmes. Man kann so leichter Probleme mit verschiedenen Browsern und sonstige Fehler beheben ohne große Teile des Programmes anpassen zu müssen.

In dem folgenden UML-Diagramm wird das Ereignis-Behandlungs-System der Online-Compiler-Benutzerschnittstelle dargestellt.

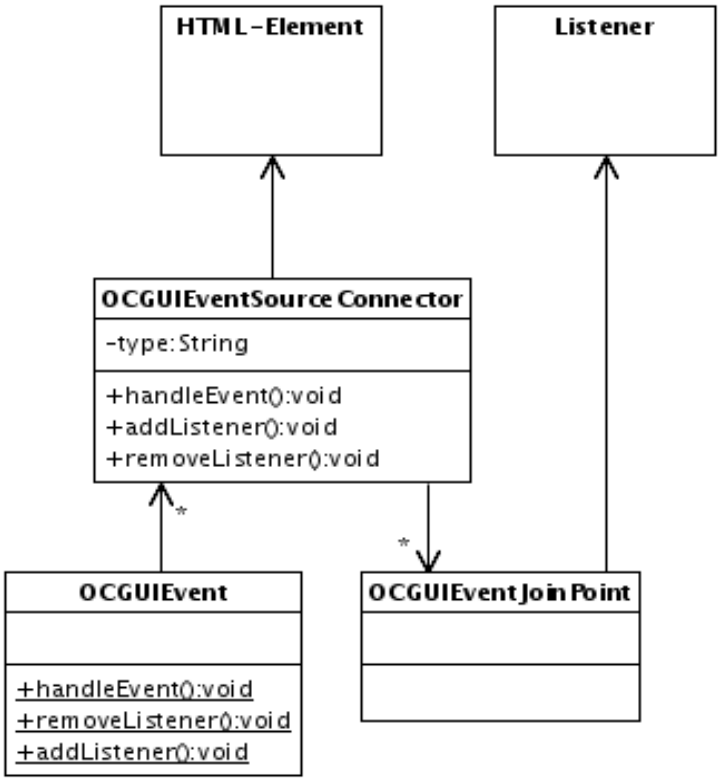


Abbildung 32: Klassendiagramm Ereignisse Webinterface

Die Klasse *HTML-Element* steht für alle HTML-Elemente, in denen Ereignisse auftreten können (zum Beispiel A-Elemente, Links in HTML-Seiten oder IMG-Elemente, Bilder). Die Klasse *Listener* steht für alle Objekte, die Ereignisse behandeln sollen, Ereignis-Handler. Die Klasse *OCGUIEvent* dient zum Erstellen, Löschen und Verwalten von Ereignissen. *OCGUIEventSourceConnector* speichert die Ereignis-Handler (*Listener*), die zu einem Ereignis eines HTML-Elementes gehören. *OCGUIEventJoinPoint* stellt schließlich die Verbindung zwischen Ereignis (*OCGUIEventSourceConnector*) und Ereignis-Handler (*Listener*) her und speichert einige Eigenschaften dieser Verbindung.

Wie oben bereits erwähnt, können keine Objekte bzw. Objekt-Methoden als Ereignis-Handler verwendet werden. Deshalb wird für jedes Ereignis die Klassen-Funktion *handleEvent* der Klasse *OCGUIEvent* als Ereignis-Handler angemeldet. Tritt ein Ereignis auf, sucht die Funktion *handleEvent* nach dem *OCGUIEventSourceConnector*-Objekt, welches für das Ereignis zuständig ist. Dieses benachrichtigt dann alle vorhandenen *Listener*-Objekte. Auf diese Weise kann man im Gegensatz zur „einfachen“ JavaScript-Ereignis-Behandlung auch Objekte als Ereignis-Handler verwenden.

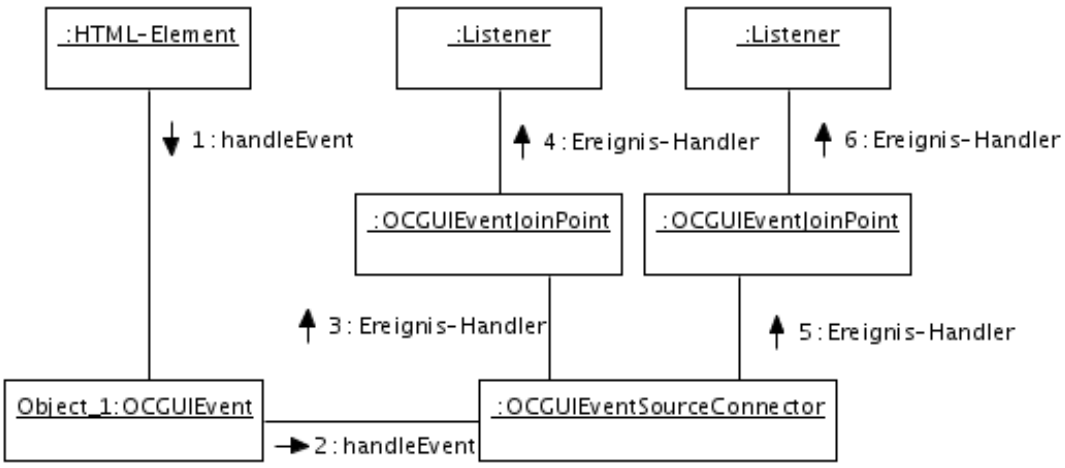


Abbildung 33: Kollaborationsdiagramm Ereignisbehandlung Webinterface

5.1.2.2.4 Quelltext-Editor

Eine der wichtigsten Komponenten der grafischen Benutzeroberfläche ist der Quelltext-Editor. Eine Komponente zum Eingeben und Bearbeiten von Quelltextdateien. Neben den Bearbeiten von Dateien müssen aber noch mehr Funktionen unterstützt werden. Dazu zählen das Hinzufügen von Debugger-Haltepunkten sowie das Hervorheben von Elementen der jeweiligen Programmiersprache wie zum Beispiel Schlüsselwörter (Syntax-Highlighting). Das einzige HTML-Element, welches das Bearbeiten von Texten unterstützt, ist das Textarea-Element. Dies hat aber den Nachteil, das man es nicht um Funktionen wie Hinzufügen von Debugger-Haltepunkten erweitern kann. Mit dem Textarea-Element kann man nur Text bearbeiten. Es ist daher ungeeignet für die Editor-Komponente der Online-Compiler-Benutzerschnittstelle.

Von einigen Web-Browsern, insbesondere neueren Versionen, wird aber seit einigen Jahren eine weitere Möglichkeit unterstützt, mit dem man Texte innerhalb eines HTML-Dokumentes bearbeiten kann: der sogenannte MIDAS-Editor *[Mozilla 1]*. Der MIDAS-Editor ist ein so genannter WYSIWYG-HTML-Editor. WYSIWYG steht für „What you see is what you get“ („Was du siehst, bekommst du auch“) und bedeutet (im Fall des MIDAS-Editors), dass man das HTML-Dokument, welches man im MIDAS-Editor bearbeitet, während der Bearbeitung schon so sieht, wie es später in einem Web-Browser angezeigt wird. Man Bearbeitet also nicht den HTML-Code des Dokumentes. Mit dem MIDAS-Editor hat man also ein Werkzeug, mit dem Dokumente bearbeitet werden können, die nicht nur Text sondern auch Bilder oder Formatierungen enthalten. Diese Eigenschaft kann man für die Editor-Komponente der Benutzerschnittstelle ausnutzen:

- Debugger-Haltepunkte werden als Bilder in die geöffnete Editor-Komponente eingefügt
- Syntax-Hervorhebungen werden durch spezielle Formatierung (Farbwechsel) durchgeführt. Zum Beispiel werden Schlüsselwörter farblich hervorgehoben.
- Der Benutzer kann den Inhalt des Editors bzw. den Quelltext der geöffneten Datei bearbeiten und verändern.

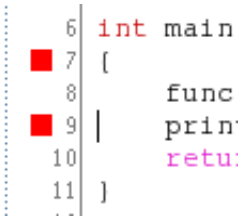
Mit dem Textarea-Element wäre nur der letzte Punkt realisierbar.

Wie bereits anfangs erwähnt, dient der MIDAS-Editor zum Bearbeiten eines HTML-Dokumentes. Der Quelltext-Editor soll aber nur Teil eines HTML-Dokumentes, der Benutzerschnittstelle, sein. Diese Problem lässt sich auf einfache Weise durch Verwendung eines IFrame-Elementes lösen. Ein IFame (Inline-Frame) ist ein HTML-Dokument, welches innerhalb eines anderen HTML-Dokumentes integriert ist. Der Quelltext-Editor wurde also in Form eines IFrames implementiert welches in die Benutzerschnittstelle (das HTML-Dokument des Online-Compilers) eingefügt werden kann. Um aus einem IFrame bzw. einem HTML-Dokument einen MIDAS-Editor zu machen, genügt ein einziger JavaScript-Befehl:

```
doc.designMode = (readOnly ? "off" : "on");
```

Die Variable *doc* ist das IFrame-Element. Wenn die Variable *readOnly* den Wert *true* hat, wird das IFrame zu einem HTML-Editor (MIDAS). Wenn die Variable den Wert *false* hat, wird aus dem IFrame wieder ein „normales“ HTML-Dokument welches nicht bearbeitet werden kann.

In der folgenden Grafik wird ein Ausschnitt des Quelltext-Editors der Online-Compiler Benutzerschnittstelle gezeigt:



Auf der linken Seite befinden sich zwei Debugger-Haltepunkte in Form von zwei roten Rechtecken. Diese können in den MIDAS-Editor wie alle anderen HTML-Elemente eingefügt werden. Die Zeilennummern sind ebenfalls eine Grafik. Auf der rechten Seite befindet sich der Quelltext, der vom Anwender bearbeitet werden kann.

Die Zeilennummern können nicht als Text in den MIDAS-Editor eingefügt werden sondern müssen als Grafik implementiert werden. Würde man die Zeilennummern als Text in den Editor einfügen, hätte der Anwender die Möglichkeit, diese zu verändern da alle Elemente im MIDAS-Editor veränderbar sind.

Zum Hervorheben von Syntax-Elemente wie Schlüsselwörtern wird der vom Anwender eingegebene Quelltext durchsucht. Wird ein Element gefunden, welches hervorgehoben werden soll, zum Beispiel ein Schlüsselwort oder ein Kommentar, wird für dieses Element eine Formatierung hinzugefügt. Dazu können beliebige HTML-Elemente wie zum Beispiel Farben (Font-Element) verwendet werden, da der MIDAS-Editor ein HTML-Editor ist und somit beliebigen HTML-Code anzeigen kann.

Um die Quelltext-Abschnitte zu finden, die hervorgehoben werden sollen, wird für jeden Dateityp (Java, Lisp, C, C++, C# und C/C++-Header-Dateien) eine Liste mit den entsprechenden Syntax-Elementen und den Format-Änderungen verwendet.

Für die Programmiersprache Java sieht diese Liste wie folgt aus:

```
/** Syntax f&uuml;r Java-Dateien. */
javaSyntax = [
      // Zeichenketten
      /([\"].*?[\"])/g,
            '<font color=\"blue\">$1</font>',
      // Zeichen
      /([\'].*?[\'])/g,
            '<font color=\"magenta\">$1</font>',
      // Schluesselwoerter
      /(\W|^)(abstract|continue|for|new|switch|assert|default|goto|
      package|synchronized|boolean|do|if|private|this|break|double|
```

```
      implements|protected|throw|byte|else|import|public|throws|case|
      enum|instanceof|return|transient|catch|extends|int|short|try|
      char|final|interface|static|void|class|finally|long|  strictfp|
      volatile|const|float|native|super|while)(\b)/g,
            '$1<font color=\"#E018CC\">$2</font>$3',
      // Datentypen
      /(\W|^)(bool|byte|char|const|double|final|float|int|long|  short|
      String|short|static|void)(\b)/g,
            '$1<font color=\"#C70609\">$2</font>$3',
      // Kommentare //
      /([^:]|^)\/\/(.*?)(<br>|<\/P>)/g,
            '$1<font color=\"green\">//$2</font>$3',
      // Kommentare /* */
      /\/\*(.*?)\*\//g,
            '<font color=\"green\">/*$1*/</font>'
];
```

Die folgende JavaScript-Zeile sorgt dafür, das „/* */“ Kommentare grün eingefärbt werden:

```
/\/\*(.*?)\*\//g, '<font color=\"green\">/*$1*/</font>'
```

Der erste Teil ist ein regulärer Ausdruck, mit dem alle Kommentare in einem Quelltext gefunden werden. Der zweite Teil ersetzt jedes gefundene Kommentar durch den HTML-Code *...* wodurch diese grün eingefärbt werden.

Neben Kommentaren werden außerdem Zeichenketten, einzelne Zeichen (Char), Schlüsselwörter und Datentypen hervorgehoben.

Der MIDAS-Editor ist ein HTML-Editor. Dass heißt, mit ihm kann man HTML-Dokumente bearbeiten. Das bedeutet, der mit dem MIDAS-Editor erstellte Text ist HTML-Code. Mit dem Quelltext-Editor soll aber kein HTML-Code, sondern Quelltexte der verschiedenen Programmiersprachen erzeugt werden. Es ist also erforderlich, aus dem HTML-Dokument welches mit dem MIDAS-Editor erstellt wurde, den tatsächlichen Quelltext herauszufiltern. Dazu reicht es aus, alle HTML-Elemente wie zum Beispiel Formatierungen (HTML-Font-Elemente) zu löschen. Übrig bleibt dann der eigentliche Quelltext. Die JavaScript-Funktion *getContent* dient zum heraus-filtern dieser HTML-Elemente:

```
getContent = function()
{
      code = this.content.innerHTML;
      code = code.replace(/<br>/gi,'\n');
      ...
      code = code.replace(/<.*?>/g,'');
      code = code.replace(/&lt;/g,'<');
      code = code.replace(/&gt;/g,'>');
      code = code.replace(/&amp;/gi,'&');
      return code;
}
```

5.1.2.2.5 Übersicht Grafische Komponenten

Neben dem Quelltext-Editor gibt es eine Reihe weiterer grafischer Komponenten. Jede Komponente wurde als JavaScript-Klasse implementiert.

JavaScript-Klasse	*Beschreibung*
OCConsole	Komponente um Ein- und Ausgabe von Programmen anzuzeigen
OCDebugger	Zeigt den Status des Debuggers sowie Debugger-Variablen an
OCFileTree	Komponente um Dateien und Verzeichnisse anzuzeigen
OCProblems	Zeigt Fehlermeldungen, Warnungen und Statusmeldungen von Compiler, Linker und Programmen an
OCMenu, OCMenuPoint, OCMenuItem	Klassen für Menüs, Popup-Menüs und Menüeinträge
OCSourceFileEditor	Der oben bereits vorgestellte Quelltext-Editor
OCGUISplitter	Komponente zum Verändern der Größen von anderen Komponenten durch „Ziehen“ mit der Maus
OCGUITabElement, OCGUITabHeader	Klassen zum Erstellen von Registerkarten
OCDocSearchDialog	Dialog um Programmiersprachen-Dokumentationen zu durchsuchen
OCLoginDialog	Dialog um Benutzer des Online-Compilers anzumelden
OCUploadDialog	Dialog zum Auswählen einer Datei um diese zum Servlet hochzuladen
OCFileArgsDialog	Dialog zum Ändern der Kommandozeilenargumente eines Programmes
OCFileTargetsDialog	Dialog zum Ändern der Linker- und Compiler-Targets einer Datei

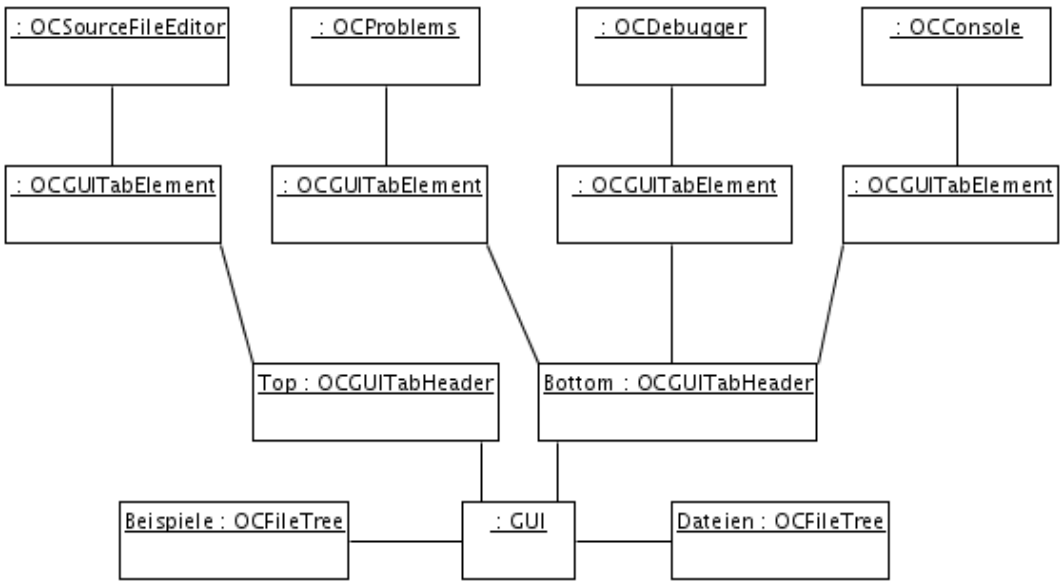


Abbildung 34: Objektdiagramm Benutzerschnittstelle Online-Compiler

5.1.2.2.6 Integration in Ilias

Der Online-Compiler muss in den Online-Kurs „Programmieren in Java“ und in den Online-Kurs „Datenstrukturen+Algorithmen“ integriert werden. Das bedeutet, das bei jedem Programm-Beispiel in den Online-Kursen ein Link vorhanden sein soll, mit dem das Programm-Beispiel im Online-Compiler geöffnet werden kann. Die einfachste und wohl auch sinnvollste Art der Integration ist, wenn über einen Link neben jedem Beispiel-Programm das Webinterface (ein geöffnetes Online-Compiler-Fenster) dazu veranlasst wird, das Beispiel-Programm zu öffnen. Durch einen Klick auf „[Beispiel-Link]“ in einen der Ilias-Kurse Online-Kurs muss also das Webinterface mit dem entsprechenden Beispiel geöffnet werden. Die einzige Möglichkeit, dem Online-Compiler das zu öffnende Beispiel mitzuteilen ist, das Beispiel in die URL zu integrieren. Ein Aufruf des Online-Compilers mithilfe einer JavaScript-Funktion wäre zwar einfacher und komfortabler, ist aber in der Regel nicht möglich da Ilias und Online-Compiler-Servlet auf verschiedenen Servern laufen werden. Ein Aufruf von JavaScript-Funktionen von verschiedenen Servern ist aber in den meisten Web-Browsern aus Sicherheitsgründen verboten.

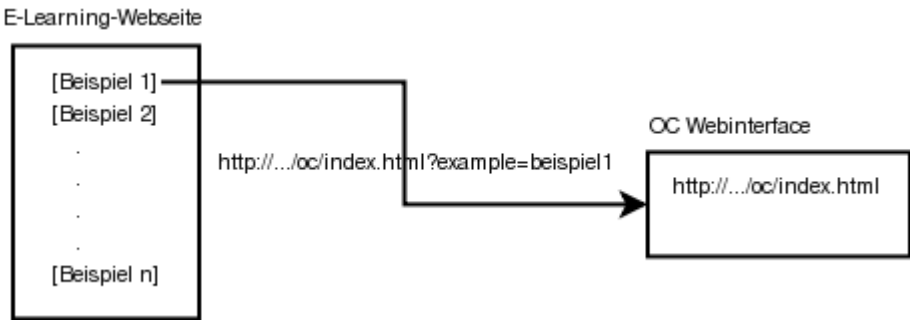


Abbildung 35: Aufruf Programm-Beispiele

Wenn ein Anwender auf den Link für ein Beispiel-Programm in einem Online-Kurs klickt, wird die entsprechende URL in einem neuen Fenster geöffnet. Das Webinterface des Online-Compilers erkennt das angegebenen Beispiel in der URL und sendet eine Anfrage zum Öffnen des Beispiels zum Servlet.

Um das Erstellen von Programm-Beispiel-Links zu vereinfachen, wurden in der Datei *include.js* einige JavaScript-Funktionen zum Aufruf des Online-Compilers erstellt. Um ein Programm-Beispiel im Online-Compiler zu öffnen, muss man lediglich die folgende Funktion in den entsprechenden Link integrieren:

```
function oc_open_example_file(example, file)
{
     try{
          var url = root + "index.html?example=" +
               encodeURIComponent(example) + "&file=" +
               encodeURIComponent(file);
          ocwindow = window.open(url, "ocwindow", "width=" + width
               + ",height=" + height);
```

```
        }catch(e){
              ocwindow = window.open(url, "ocwindow", "width=" + width
              + ",height=" + height);
        };
        ocwindow.focus();
}
```

Neben den Aufruf von Beispielen soll der Online-Compiler auch noch direkt in einem Frame-Fenster in den Online-Kurs integriert werden. Dazu wurde eine spezielle „kleine“ Version der Online-Compiler-Benutzerschnittstelle entwickelt. Diese Version unterscheidet sich in drei Punkten von der „normalen“ Version des Online-Compilers:

- die Komponenten zum Anzeigen von Dateien und Beispielen (Datei-Baum) wurde entfernt da, nur ein einziges Beispiel-Programm angezeigt werden soll
- der Menüpunkt „NEUE DATEI“ wurde entfernt, da keine neuen Datei erzeugt werden sollen sondern nur ein einziges Beispiel angezeigt werden soll
- Statt einem Editor gibt es mehrere in denen alle Dateien des geöffneten Programm-Beispiels angezeigt werden

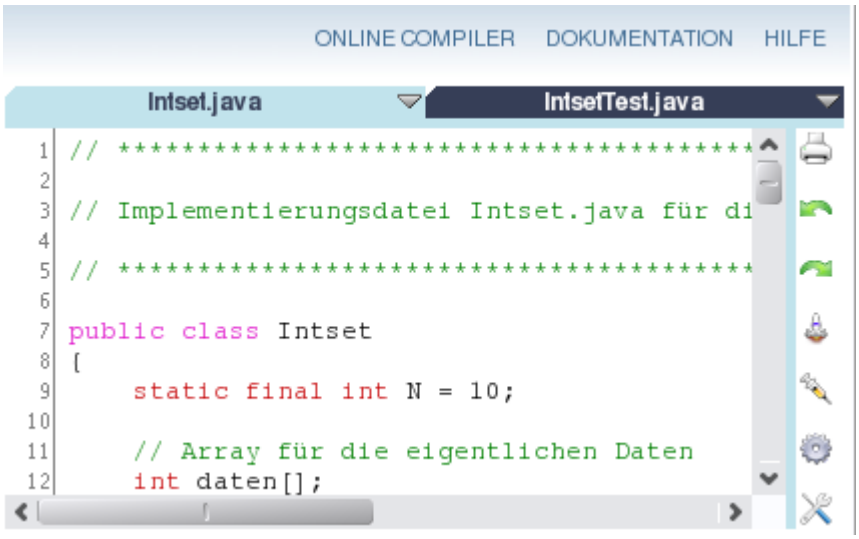


Abbildung 36: Ilias-Version der Benutzerschnittstelle

Um für jede Datei eines Beispiels einen Editor zu erzeugen, muss lediglich für jede Datei eine neue Instanz der Klasse *OCSourceFileEditor* erzeugt werden und diese in eine neue Instanz der Klasse *OCGUITabElement* (Registerkarte) eingefügt werden:

```
OCGUI._newFileEditor = function(title, file)
{
      var fileEditor = new Object();
      fileEditor.editor = new OCSourceFileEditor("main_file_view");
```

```
        fileEditor.title = title;
        fileEditor.file = file;
        fileEditor.name = "file_editor_" + this.fileEditorList.length;
        this.fileEditorList.push(fileEditor);
        new OCGUITabElement(this.mainTabPanel, fileEditor.name,
             title, "top", fileEditor.editor);
}
```

Die „kleine“ Version des Online-Compilers kann genauso wie die „normale“ Version über die URL aufgerufen werden. Der einzige Unterschied ist, das man statt „index.html“ in der URL den Namen „indexklein.html“ verwenden muss.

5.1.2.2.7 Kommunikation mit dem Servlet / MVC

Die folgende Grafik stellt die Ablauf der Kommunikation zwischen Webinterface (Online-Compiler-Client) und Servlet dar.

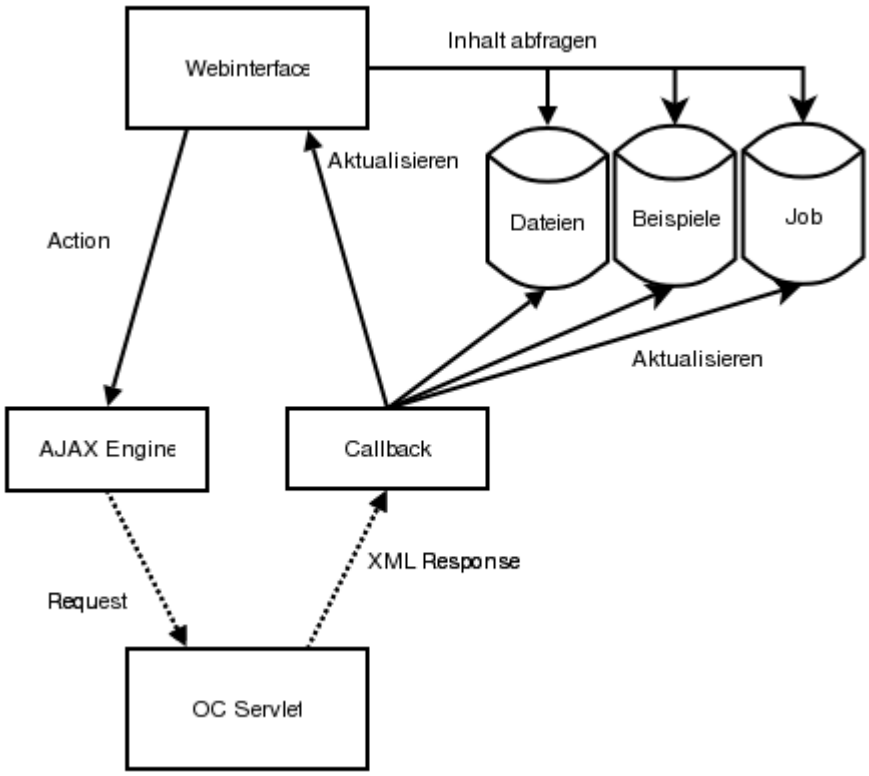


Abbildung 37: Ajax Kommunikation Client-Server

Wird im Webinterface vom Benutzer ein Befehl (Action) aufgerufen (z.B. ein Klick auf einen Menüpunkt), bewirkt dies in der Regel, dass ein Request über die sogenannte AJAX-Engine zum Online-Compiler-Servlet gesendet wird. Bei einer Antwort des Servlets wird eine Callback-Methode angestoßen, welche die Antwort bearbeitet. Zum Beispiel wird dann der Dateibaum (die Dateien und Verzeichnisse, die der Anwender erstellt hat) aktualisiert. Außerdem werden die Inhalte der

HTML-Seite aktualisiert.

Das Webinterface speichert Informationen über Dateien, Beispiele und den laufenden Job. Falls sich eine Änderung ergibt, wird diese Änderung vom Servlet zum Webinterface gesendet und dort an der entsprechenden Stelle eingetragen. Client und Servlet arbeiten also nach dem MVC-Muster:

- Änderungen am **Modell** (z.B. Dateien) werden vom Servlet an den Client übertragen
- die Modelldaten werden im **View**, dem Client angezeigt
- eine Action (**Controller**) wird zum Servlet übertragen und bewirkt eine Modelländerung

Damit nicht ständig alle Modell-Daten vom Servlet übertragen werden müssen, werden die wichtigsten Daten der Dateien, Beispiel und des laufenden Jobs im Webinterface zwischengespeichert.

Das wichtigste Element im Webinterface ist die AJAX-Engine, die die Kommunikation mit dem OC-Servlet erst möglich macht. Dazu wurde ein **AJAX-Framework** (Datei *ace.js*), welches von Li Shen entwickelt wurde, verwendet *[Li Shen 2006]*. Das AJAX-Framework ist eine einfache JavaScript Erweiterung für Browser mit der man relativ einfach XML-Requests an XML-Server senden kann.

Wenn ein XML-Response, eine Antwort des Servlets, bei einem Client ankommt, wird eine sogenannte Callback-Funktion aufgerufen die den XML-Inhalt des Servlet-Response analysiert und entsprechende Aufgaben (z.B. aktualisieren des Webinterface) ausführt. Um den XML-Inhalt von einem Servlet-Response zu analysieren wird ein XML-Parser verwendet *[JSXML]*.

Beispiel für eine für eine Servlet-Anfrage:

```
OCGUI.actionLogout = function()
{
     if(!this.loggedIn)
          return;
     var request = new Ace.Request(this,
          function(response){
               if(!this.handleResponse) return;
               if(response.success != true)
                    this._showServerErrorMessage(response.success,
                    "logout", response.lastCommand.name);
               else
                    this._logout(); // Abmelden abschliessen
            }
       );
     if(!this.jobRequest && !this.jobRunning)
       this._ajaxRequestAddSaveFileCommandForAllEditors(request);
     request.addCommand("<logout />");
```

```
        request.finish();
        this.ajaxConnection.invoke(request);
}
```

Das obige Beispiel dient zum Abmelden eines Anwenders vom Online-Compiler. Das Objekt der Klasse *Ace.Request* ist die eigentliche Anfrage die von der Ajax-Engine in XML-Code umgewandelt und zum Servlet gesendet wird. Das Objekt hat im Konstruktor bereits die Callback-Funktion, also die Funktion, die bei einer Servlet-Antwort ausgeführt wird. Mit dem Befehl *addCommand* wird ein Befehl des OC-Kommunikationsprotokoll zur Anfrage hinzugefügt. In diesem Fall der *logout*-Befehl zum Abmelden des Anwenders. Vorher wird mit der Funktion *_ajaxRequestAddSaveFileCommandForAllEditors* für jede geöffnete Datei ein Befehl zum Speichern der Datei zu der Anfrage hinzugefügt. Auf diese Weise werden vor dem Abmelden eines Anwenders seine Dateien gesichert.

Wie man in dem obigen Beispiel sehen kann, sind in der Callback-Funktion keine Befehle zum analysieren des XML-Codes der Servlet-Antwort enthalten. Da die Analyse des XML-Codes bei jeder Antwort durchgeführt werden muss, wurden dazu einige JavaScript-Funktionen erstellt die automatisch vor jeder Callback-Funktion aufgerufen werden. Auf diese Weise spart man Platz und man kann sich bei den Callback-Funktionen auf das Wesentliche konzentrieren. Die Funktionen, die vor dem Aufruf einer Callback-Funktion ausgeführt werden sind:

- **preparateAjaxResponseCommands:**

  Analysiert den XML-Code einer Servlet-Antwort und wandelt ihn in JavaScript-Objekte um. Auf diese Weise muss man nicht jedesmal, wenn man auf Informationen aus der Servlet-Antwort zugreifen möchte, den XML-Code durchlaufen sondern kann einfach auf die Objekt-Attribute zugreifen. So ist zum Beispiel das Attribut *response.success* genau dann *true*, wenn kein Fehler in der Servlet-Antwort enthalten ist.

- **_handleAjaxEvents:**

  Verarbeitet alle Modell-Änderungen von Dateien und Beispiel-Programmen

- **_handleJobEvents:**

  Verarbeitet alle Modell-Änderungen von dem laufenden Job

5.2 Multi-Task-Java

Ziel dieses Kapitels ist die Beschreibung von Implementierungsdetails der MJVM (Multi-Task-Java-VM).

Neben diesem Kapitel steht auch noch eine ausführliche **Inline-Dokumentation** in den Java Quelldateien (Javadoc) zur Verfügung. Diese befindet sich auf der CD im Verzeichnis `doc/api/MultiTaskJavaVM` oder kann Online über die URL http://javock.fh-trier.de:8080/oc/mjvm/api/index.html abgerufen werden.

5.2.1 Datei-Übersicht

Die Quelldateien der MJVM befinden sich im Verzeichnis `source/MultiTastJavaVM`. Die MJVM besteht aus folgenden Java-Paketen:

Paketname	*Inhalt*
mjvm.server.io	Klassen für Ein- und Ausgabe von Java-Anwendungen, die in einer MJVM ausgeführt werden.
mjvm.server.process	Klassen für Java-Anwendungen, die in einer MJVM ausgeführt werden.
mjvm.server	Hauptklassen für eine MJVM (Java-Anwendung).
mjvm.exceptions	Paket welches die Exceptions der MJVM enthält.
mjvm.connector	Klassen für MJVM-Clients.
mjvm.connector.process	Klassen für MJVM-Clients.
mjvm.connector.demo	Ein Programm zum Testen der MJVM (Beispiel eines MJVM-Clients).
java.lang	An die MJVM angepasste Klassen des java.lang-Paketes (System-Klassen).
java.io	An die MJVM angepasste Klassen des java.io-Paketes (System-Klassen).

5.2.2 Compilierung der MJVM

Die MJVM ist vollständig in Java Programmiert worden. Um die MJVM zu Compilieren, müssen alle Quelltextdateien im Verzeichnis `source/MultiTaskJavaVM` compiliert werden.

5.2.3 Demo-Programm ausführen

Das Verzeichnis *source/MultiTaskJavaVM* enthält ein Beispiel-Programm welches eine MJVM zum Ausführen eines Java-Programmes verwendet. Das Programm befindet sich in der Klasse *mjvm.connector.demo.DemoFrontend*. Das Beispiel-Programm stellt eine Verbindung zu einer laufenden MJVM her und führt auf der MJVM ein Java-Programm aus. Um die Geschwindigkeit mit einer herkömmlichen Java-VM zu vergleichen, wird das Java-Programm auch mit einer „normalen“ Java-VM von SUN ausgeführt. Das Beispiel-Programm ist die Klasse *HelloWorld* im Verzeichnis *source/MultiTaskJavaVM*.

Bevor das Demo-Programm ausgeführt werden kann, muss eine MJVM gestartet werden. Dazu muss der folgende Befehl aufgerufen werden:

```
java -cp .
     -Xbootclasspath/p:.
     -Xbootclasspath/a:/usr/lib/java/lib/tools.jar:.
     mjvm.server.MJVMVirtualMachine
```

Mit dem Befehl wird eine MJVM gestartet. Der Befehl muss in dem Verzeichnis ausgeführt werden, in dem sich die Klassen der MJVM befinden. Der Pfad */usr/lib/java/lib/tools.jar* gibt die JAR-Datei an, in der sich die JDI-Klassen von SUN befinden.

Nachdem die MJVM gestartet wurde, kann man das Demo-Programm mit dem folgenden Befehl starten:

```
java mjvm.connector.demo.DemoFrontend 50 ./rt HalloWelt
```

Das Demo-Programm hat 3 Kommandozeilen-Parameter:

- Die Anzahl der Instanzen, die von dem Beispiel-Programm ausgeführt werden sollen. Bei 50 wird 50 mal das Beispiel-Programm in der MJVM ausgeführt.
- Der zweite Parameter gibt das Verzeichnis an, in der sich das Beispiel-Programm befindet.
- Der dritte Parameter gibt den Klassen-Namen des Beispiel-Programms an.

5.2.4 Implementierung der MJVM

5.2.4.1 Kommunikation zwischen Client-Anwendungen und einer MJVM

Die MJVM (Multi-Task-Java-VM-Anwendung) soll mit anderen Java-Anwendungen (Clients) wie dem Online-Compiler gesteuert werden können. Das heißt, Client und MJVM müssen miteinander kommunizieren. Dazu wird RMI, Remote Method Invokation, verwendet *[Sun 5]*. RMI ist ein Protokoll zum Aufruf von Methoden über Java-Anwendungs-Grenzen hinaus. Das heißt, mit RMI kann man Methoden in einer anderen Java-Anwendung aufrufen. Beide Anwendungen können dabei auch auf verschiedenen Computern laufen. RMI hat den Vorteil, dass es speziell für Java entwickelt wurde. Der Aufruf von Methoden in anderen Java-Anwendungen unterscheidet sich mit RMI kaum von dem Aufruf lokaler Methoden.

Bei RMI gibt es immer eine Server-Anwendung und eine Client-Anwendung wobei eine Java-Anwendung nicht nur Server oder nur Client sein muss sondern auch beides sein kann. Die Server-Anwendung ist der Besitzer eines Objektes, dem sogenannten Remote-Objekt. Das Remote-Objekt bzw. deren Klasse muss eine Schnittstelle implementieren, die Remote-Schnittstelle. Die Client-Anwendung kann auf die Methoden des Remote-Objektes zugreifen, welche in der Remote-Schnittstelle deklariert sind. RMI übernimmt automatisch das Übertragen der Daten. Tritt bei der Kommunikation zwischen Client und Server ein Fehler auf, wird eine *RemoteException* ausgelöst.

Eine weitere Funktion von RMI ist der sogenannte Naming-Service. Ein RMI-Server, also eine Anwendung, die ein Remote-Objekt besitzt, kann an einem bestimmten TCP-Port auf Client-Anfragen warten. Client-Anfragen werden durch das Senden der Remote-Schnittstelle beantwortet. Der RMI-Client kann dann über die Methoden der Remote-Schnittstelle auf die RMI-Server Funktionen zugreifen. Für die MJVM bedeutet dies, das die MJVM auf Clients wartet, die Java-Programme in der MJVM ausführen möchten.

Remote-Schnittstellen der MJVM (Schnittstellen der Remote-Objekte, auf die ein Client zugreifen kann):

- *MJVMEnrolInterface*: dient zum Herstellen der Verbindung zwischen einer MJVM und den Clients (das Remote-Objekt, welches über den RMI-Naming-Service angesprochen werden kann um die Verbindung zur MJVM herzustellen)
- *MJVMServerInterface*: die Methoden der MJVM (z.B. Starten einer neuen Java-Anwendung)
- *MJVMProcessServerInterface*: zur Kontrolle einer Java-Anwendung, die in der MJVM ausgeführt wird

Remote-Schnittstellen eines MJVM-Clients (Schnittstellen der Remote-Objekte, auf die die MJVM zugreifen kann um zum Beispiel Ereignisse dem Client zu melden):

- *MJVMClientInterface*: Methoden des Clients, die die MJVM aufrufen kann
- *MJVMProcessClientInterface*: Methoden des Clients, die im Zusammenhang mit einer Java-Anwendung, die von der MJVM ausgeführt wird, aufgerufen werden können

Die folgende Abbildung verdeutlicht die Funktionsweise von RMI zum Austausch von Befehlen und Ereignissen von Java-Programmen, die in einer MJVM ausgeführt werden:

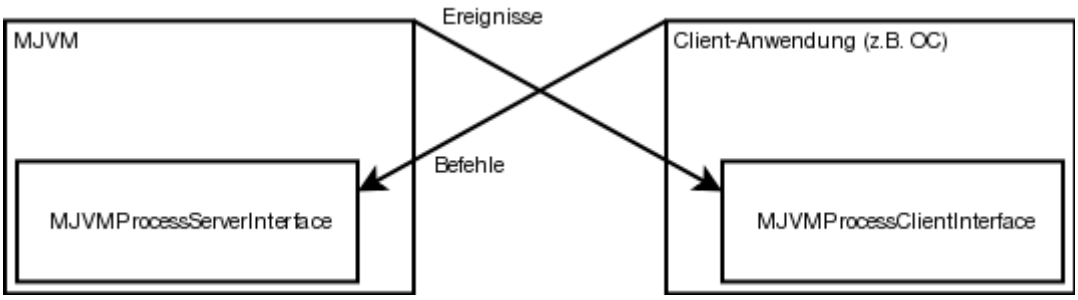


Abbildung 38: Kommunikation MJVM und Client-Anwendung

Die Client-Anwendung (zum Beispiel der Online-Compiler) besitzt ein Remote-Objekt der Schnittstelle *MJVMProcessClientInterface* auf das die MJVM zugreifen kann. Tritt ein Ereignis auf (zum Beispiel wenn das Java-Programm beendet ist), wird dieses Ereignis dem Remote-Objekt der Client-Anwendung mitgeteilt. Die MJVM hat selbst auch ein Remote-Objekt der Schnittstelle *MJVMProcessServerInterface*. An dieses Objekt sendet der Client durch Methodenaufrufe Befehle für die Java-Anwendung (zum Beispiel zum Stoppen der Java-Anwendung). Beide Programme sind also gleichzeitig RMI-Client und RMI-Server.

5.2.4.1.1 Verbindung zu einer MJVM herstellen

Wie bereits oben erwähnt wurde, wartet die MJVM an einem speziellem Port auf Client-Verbindungen. Wenn ein Client eine Anfrage über RMI stellt, wird eine Instanz der Schnittstelle *MJVMEnrolInterface* zurückgeliefert. Die Methode *startRMI* der Klasse *mjvm.connector.MJVMConnector* stellt die Verbindung zu einer laufenden MJVM her.

```
private void startRMI()
     throws MJVMException
{
     try{
          System.out.println("Start RMI");
          MJVMServerEnrolInterface enrol =
               (MJVMServerEnrolInterface)Naming.lookup("rmi://" +
               mjvmHost + ":" + mjvmPort + "/" + mjvmServiceName);
```

```
            server = enrol.login(this);
            System.out.println("Start Finished, MJVM Ready");
      }catch(Exception e){
            throw new MJVMException("Error while launch RMI", e);
      }
}
```

Die Methode *naming.lookup* liefert das *MJVMEnrolInterace*-Objekt der MJVM zurück. Die Methode *login* dieses Remote-Objektes liefert schließlich ein Remote-Objekt der Schnittstelle *MJVMServerInterface* zurück, mit dem auf Funktionen der MJVM wie „Start einer Java-Anwendung“ zugegriffen werden kann.

Der RMI-Service der MJVM wird mit der Methode *startRMISevice* der Klasse *MJVMVirtualMachine* gestartet:

```
private static void startRMIService(int port, String url)
      throws RemoteException, MalformedURLException
{
      try{
            Registry registry = LocateRegistry.getRegistry(port);
            registry.list();
      }catch(RemoteException e){
            Registry registry = LocateRegistry.createRegistry(port);
      }
      Naming.rebind(url, vm);
}
```

5.2.4.2 Ausführen von Java-Anwendungen

In dem folgenden Klassendiagramm werden die Klassen, die für eine Java-Anwendung, die in einer MJVM ausgeführt werden soll, benötigt werden, dargestellt.

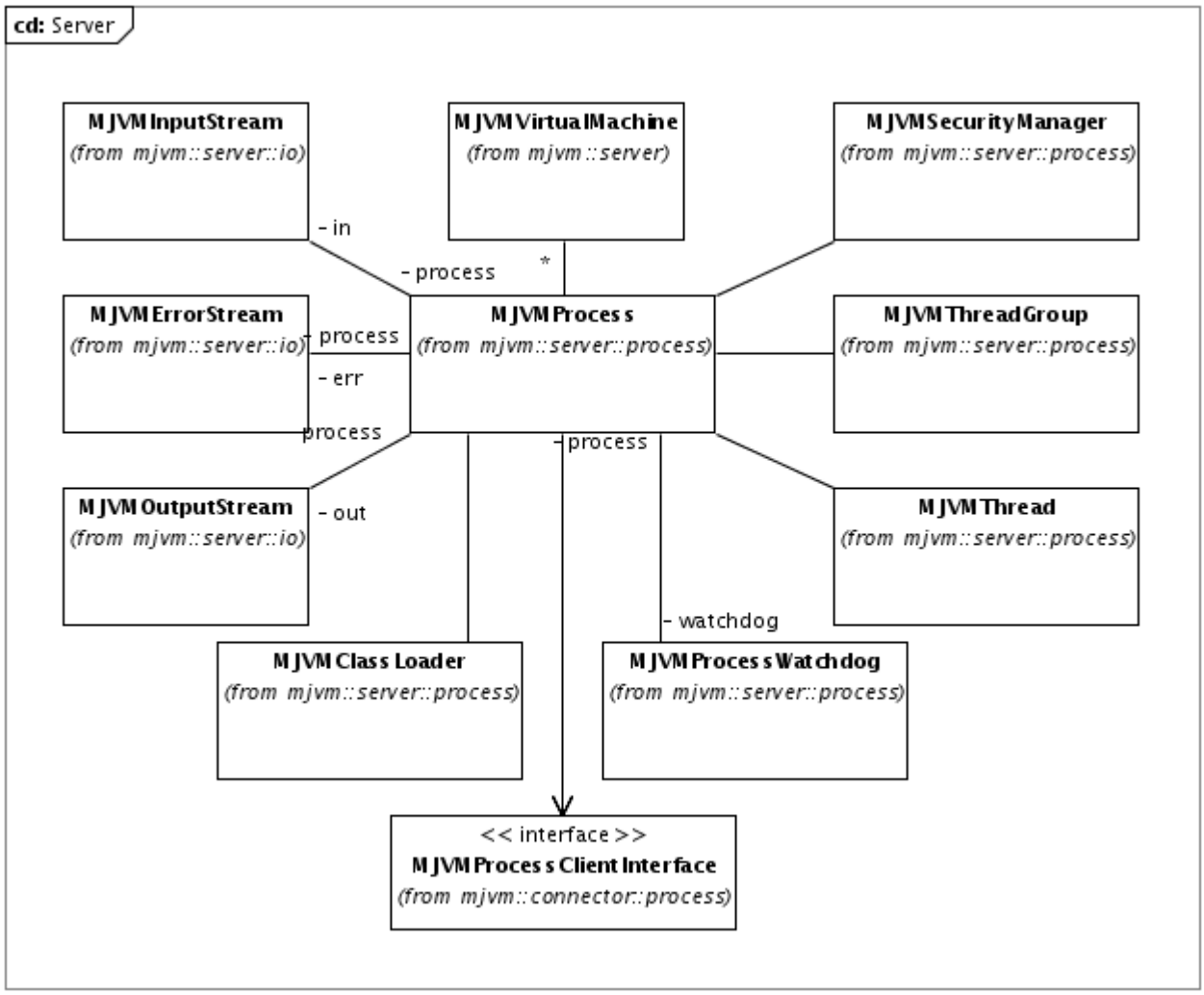


Abbildung 39: Klassenhierarchie MJVM-Prozess

Die Klasse *MJVMProcess* erweitert die Remote-Schnittstelle *MJVMProcessClientInterface*. Auf diese Weise kann das Client-Programm, welches die Java-Anwendung gestartet hat, die Java-Anwendung über RMI steuern. Die Klassen *MJVMInputStream*, *MJVMErrorStream* und *MJVMOutputStream* dienen als Ein- und Ausgabeströme der Java-Anwendung. Da mit einer MJVM mehrere Java-Anwendungen parallel ausgeführt werden, kann man für die Ein- und Ausgabe nicht die sonst üblichen Datenströme der *System*-Klasse verwenden sondern muss für jede Java-Anwendung bzw. jedes *MJVMProcess*-Objekt eigene Ein- und Ausgabeströme verwenden.

Das wichtigste einer Java-Anwendung die in einer MJVM ausgeführt werden ist ihr eigener Thread (Klasse *MJVMThread*). Damit in einer Java-VM (Virtuall Machine) mehrere Java-Anwendungen parallel ausgeführt werden, muss für jede Anwendung ein eigener Thread verwendet werden.

Das größte Problem, welches bei der Implementierung der MJVM berücksichtigt werden muss ist, das die Java-Programme unabhängig voneinander laufen müssen. Dass heißt, es darf zum Beispiel nicht möglich sein, das ein Java-Programm den Thread eines anderen Java-Programms stoppt. Aus diesem Grunde sind für jedes Java-Programm (oder jedes *MJVMProcess*-Objekt) folgende Dinge notwendig:

- wie oben bereits erwähnt eigene **Ein**- und **Ausgabe-Ströme**
- eine eigene **Thread-Gruppe** (Klasse *MJVMThreadGroup*):

  Threads werden in Java in Gruppen zusammengefasst. Ein Thread darf nur auf Threads zugreifen, die sich in der gleichen Gruppe befinden. Normalerweise laufen alle Threads einer JVM in einer Gruppe, da es ja auch nur ein Java-Programm gibt, welches ausgeführt wird. Bei der MJVM wird hingegen für jedes Java-Programm eine eigene Thread-Gruppe benötigt. Auf diese Weise kann man nicht auf Threads zugreifen, die zu einer anderen Gruppe bzw. einer anderen Java-Anwendung gehören.
- einen eigenen **Security-Manager** (Klasse *MJVMSecurityManager*):

  Wenn der Online-Compiler Java-Programme ausführt, müssen diese mit dem Java-Security-Manager überwacht werden. Weil jedes Java-Programm andere Rechte hat (jedes Programm kann zu einem anderen Benutzer gehören), kann man bei der MJVM nicht nur einen Java-Security-Manager verwenden, sondern muss für jede Java-Anwendung einen eigenen erstellen.
- einen Thread, der die **Laufzeit der Java-Anwendung** überwacht (Klasse *MJVMProcessWatchdog*):

  Anwender-Programme die vom Online-Compiler ausgeführt werden haben eine maximale Laufzeit. Wird diese Zeit von einem Programm überschritten, wird es beendet (Timeout). Die MJVM muss also auch die Laufzeit der einzelnen Programme überwachen. Dazu dient ein eigener Thread. Der Watchdog-Thread.
- eine **Remote-Schnittstelle** damit Client-Programme der MJVM (zum Beispiel der Online-Compiler) einzelne Java-Programme steuern können (Klasse *MJVMProcessClientInterface*)
- einen eigenen **Class-Loader** (Klasse *MJVMClassLoader*):

  Ein Class-Loader ist eine Klasse, die andere Klassen aus den Class-Dateien lädt, wenn sie benötigt werden. Es gibt bei Java zwei Class-Loader: der System-Class-Loader der Klassen der Java-Runtime-Environment (JRE) lädt, und der User-Class-Loader, der die Klassen des Java-Programmes lädt. Ein eigener Class-Loader (User-Class-Loader) ist für jede Java-Anwendung, die in der MJVM ausgeführt wird, notwendig weil jede Java-Anwendung auch eigene Classpath-Werte verwendet. Eine weiterer Grund für eigene Class-Loader für alle Java-Anwendungen, die in der MJVM ausgeführt werden, sind die static-Methoden und static-Attribute. Wenn zwei Java-Anwendungen die Klasse X verwenden und in dieser Klasse auf das static-Attribut Y zugreifen, müssen für beide Anwendungen unterschiedliche Werte des Attributs gespeichert werden. Mit unterschiedlichen Class-Loadern kann man auch unterschiedliche Instanzen (Klassen-Definitionen) der selben Klasse in einer JVM gleichzeitig verwenden. Jede Java-Anwendung hat dadurch eigene static-Attribute und static-Methoden der selben Klasse (bzw. der selben Class-Datei).

Der Konstruktor der Klasse MJVMThread:

```
public MJVMThread(MJVMProcess process)
{
      super(process.getThreadGroup(), process.getName() +
            "_main_thread");
      this.process = process;
      setDaemon(false);
      setContextClassLoader(process.getClassLoader());
      try{
            mainClass = Class.forName(process.getMainClass(), true,
                  process.getClassLoader());
      }catch(ClassNotFoundException e){}
}
```

Im Konstruktor wird die Thread-Gruppe des *MJVMThreads* festgelegt. Das gleiche gilt für den Class-Loader mit dem Befehl *setContextClassLoader*. Alle Threads, die von dem *MJVMThread* abgeleitet werden, verwenden dann den gleichen Class-Loader und die gleiche Thread-Gruppe.

Run-Methode des MJVM-Threads:

```
public void run()
{
      try{
            if(mainClass == null)
                  throw new
                  ClassNotFoundException(process.getMainClass());
            Class[] params = new Class[1];
            params[0] = process.getArgs().getClass();
            Object[] mparams = new Object[1];
            mparams[0] = process.getArgs();
            mainClass.getMethod("main", params).invoke(null,mparams);
      }catch(Exception e){
            e.printStackTrace();
            System.exit(-1);
      }
}
```

Die einzige Methode, die ein *MJVMThread* ausführen muss, ist die *main*-Methode des Java-Programmes. Dazu wird mit dem Befehl *getMethod* die *main*-Methode in der Haupt-Klasse (die Klasse, die die *main*-Methode enthält) gesucht und mit *invoke* aufgerufen. Das Ende des *MJVMThreads* bedeutet aber nicht unbedingt das Ende der Java-Anwendung. Für jede Java-Anwendung gilt: die Anwendung ist dann zu ende, wenn alle (nicht-Dämon-)Threads beendet sind. Das heißt, um auf das Ende der Java-Anwendung zu warten, muss gewartet werden, bis alle Threads beendet sind. Dazu dient die Methode *waitForThreads* der Klasse *MJVMThreadGroup*:

```
public boolean waitForThreads(long time)
{
     try{
          long startTime = System.currentTimeMillis();
          while(true){
               Thread[] threads;
               boolean onlyDaemons = true;
               synchronized(this){
                    threads = new Thread[activeCount() + 10];
                    enumerate(threads, true);
                    for(int i = 0; i < threads.length; i++)
                         if(threads[i] != null && !
                              threads[i].isDaemon()){
                              onlyDaemons = false;
                              break;
                         }
               }
               if(onlyDaemons)
                    return true;
               else
                    for(int i = 0; i < threads.length; i++){
                         long restTime = time -
                              (System.currentTimeMillis() -
                              startTime);
                         if(threads[i] != null && !
                              threads[i].isDaemon() &&
                              threads[i].isAlive()){
                              if(time > 0 && restTime <= 0)
                                   return false;
                              if(time > 0)
                                   threads[i].join(restTime);
                              else
                                   threads[i].join();
                         }
                    }
          }
     }catch(Exception e){
          e.printStackTrace();
          return false;
}
```

5.2.4.3 Die Klasse MJVMVirtualMachine

In dem unteren Klassendiagramm werden die restlichen Klassen der MJVM dargestellt. Diese Klassen dienen zum Steuern der MJVM selbst (also des MJVM-Programms und nicht der Java-Programme, die in der MJVM ausgeführt werden).

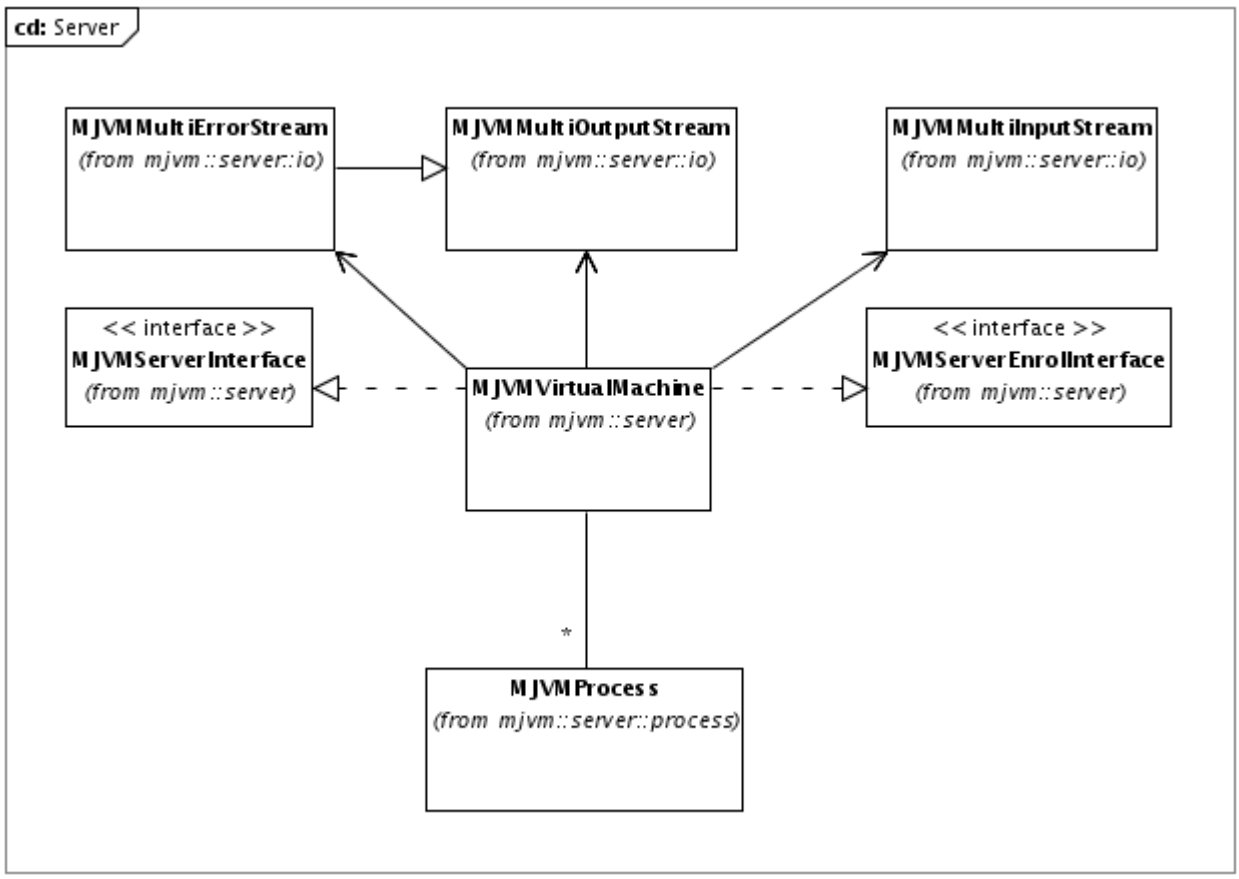


Abbildung 40: Klassenhierarchie MJVM (Java-Anwendung)

Die Klasse *MJVMVirtualMachine* ist das eigentliche MJVM-Programm. Die *main*-Methode der Klasse *MJVMVirtualMachine* startet den RMI-Service, damit Clients eine Verbindung zur MJVM herstellen können. Ein RMI-Service arbeitet immer als eigenständiger Thread. Das MJVM-Programm (bzw. der RMI-Service) wird also nicht nach Ausführung der *main*-Methode beendet.

```
public static void main(String[] args)
{
     try{
          String serviceName = MJVMConnector.DEFAULT_MJVM_SERVICE;
          int port = MJVMConnector.DEFAULT_MJVM_PORT;
          try{
               serviceName = args[0];
               port = Integer.parseInt(args[1]);
          }catch(Exception e){}
          initVM(port, serviceName);
```

```
        }catch(Exception e){
            e.printStackTrace();
        }
}
```

Die Klasse *MJVMVirtualMachine* implementiert die beiden Remote-Schnittstellen *MJVMServerInterface* und *MJVMServerEnrolInterface*. Die Schnittstelle *MJVMServerEnrolInterface* hat nur eine Methode: die Methode *login* mit der Client-Anwendungen eine Verbindung zur MJVM herstellen können. Die Schnittstelle *MJVMServerInterface* hat zwei Methoden:

```
public MJVMProcessServerInterface createProcess(String mainClass,
        String[] classPath, String workingDir, long waitTime,
        String[] args)
        throws RemoteException;

public void killAll()
        throws RemoteException;
```

Die Methode *createProcess* erstellt eine neue Java-Anwendung in der MJVM. Als Parameter muss man den Namen der Klasse, welche die *main*-Methode des Programms enthält, angeben. Weitere Parameter sind der Class-Path, das Arbeitsverzeichnis, die maximale Laufzeit des Programms und die Kommandozeilenargumente. Das Ergebnis der Methode ist eine Instanz der Remote-Schnittstelle *MJVMProcessServerInterface* mit der über RMI die Java-Anwendung gesteuert werden kann. Die Methode *killAll* beendet die MJVM und damit auch alle Java-Anwendungen, die in der MJVM ausgeführt werden.

5.2.4.3.1 System-Klassen

Die System-Klassen (Klassen der Java-Runtime-Environment) werden vom sogenannten System-Class-Loader vor dem eigentlichen Ausführen des MJVM-Programms geladen. Das heißt, man kann von diesen Klassen nicht mit dem *MJVMClassLoader* eigene Klassen-Definitionen für die einzelnen Java-Anwendungen laden. Mit anderen Worten: alle Java-Anwendungen die in einer MJVM ausgeführt werden, greifen auf die gleichen static-Attribute und static-Methoden der System-Klassen zu. Die bedeutet zum Beispiel, das alle in den gleichen Ausgabe-Strom, das Attribut *out* der Klasse *System,* schreiben. Damit alle Java-Anwendungen dennoch auf eigene Attribute und Methoden zugreifen und sich die Java-Anwendungen nicht gegenseitig stören können, müssen alle static-Attribute und static-Methoden der Java-Runtime-Environment durch Methoden und Attribute ersetzt werden, die einen Zugriff durch mehrere Java-Anwendungen erlauben, ohne das sich die Java-Anwendungen gegenseitig stören. Dazu müssen einige Klassen der Java-Runtime-Environment überschrieben werden. In den Paketen *java.io* und *java.lang* befinden sich einige dieser Klassen, die als Ersatz für die Standard-Klassen der JRE dienen.

Als Beispiel für eine an MJVM angepasste Methode dient die Methode *exit* der Klasse *Runtime* die das Beenden von Java-Anwendungen ausführt:

```
public void exit(int status){
     SecurityManager security = System.getSecurityManager();
     if(security != null){
     security.checkExit(status);
     }
     //**********Begin Änderungen MJVM**********//
     MJVMProcess process =
          MJVMVirtualMachine.getProcessByThread(
               Thread.currentThread());
     if(process != null){
          process.setExitCode(status);
          process.getThreadGroup().killAll();
          process.getOut().flush();
          process.getOut().close();
          process.getErr().flush();
          process.getErr().close();
          Thread.currentThread().stop();
          return;
     }
     //**********Ende Änderungen MJVM**********//
     Shutdown.exit(status);
}
```

Die Methode *getProcessByThread* ermittelt die Java-Anwendung (Objekt der Klasse *MJVMProcess*), zu der der Thread gehört, der die *exit*-Methode aufgerufen hat. Falls eine Java-Anwendung gefunden wurde, werden alle Threads der *MJVMThreadGroup* der Java-Anwendung mit dem Befehl *killAll* beendet. Falls keine Java-Anwendung gefunden wurde, wurde die *exit*-Methode von einem anderen Thread aufgerufen und die gesamte MJVM wird beendet.

Bei jeder static-Methode muss also zunächst die Java-Anwendung ermittelt werden, die die Methode aufgerufen hat. Dazu wird die Methode *getProcessByThread* verwendet.

Die Klassen *MJVMMultiErrorStream*, *MJVMMultiOutputStream* und *MJVMMultiInputStream* dienen als Ersatz für die static-Attribute *err*, *out* und *in* der System-Klasse. Diese Klassen sind einfache Ein- und Ausgabe-Streams mit dem Unterschied, dass bei jedem Methodenaufruf die Java-Anwendung, welche die Methode aufgerufen hat, ermittelt wird (Methode *getProcessByThread*). Wenn eine Java-Anwendung gefunden wurde, wird der Methodenaufruf an den Ein- beziehungsweise Ausgabe-Stream der Java-Anwendung (Klassen *MJVMErrorStream*, *MJVMOutputStream* und *MJVMInputStream*) weitergeleitet. Auf diese Weise greifen zwar alle Java-Anwendungen auf die gleichen static-Attribute zu (*System.err*, *System.out* und *System.in*), haben aber trotzdem unabhängig voneinander eine eigene Ein- und Ausgabe.

5.2.4.4 MJVM-Client Klassen

Die Klassen des Paketes *mjvm.connector* haben die Aufgabe, eine Client-Anwendung (zum Beispiel den Online-Compiler) um Funktionen zum Ausführen und Steuern von Java-Programmen in einer MJVM zu erweitern. Im folgenden Klassendiagramm werden alle Klassen des Paketes *mjvm.connector* dargestellt.

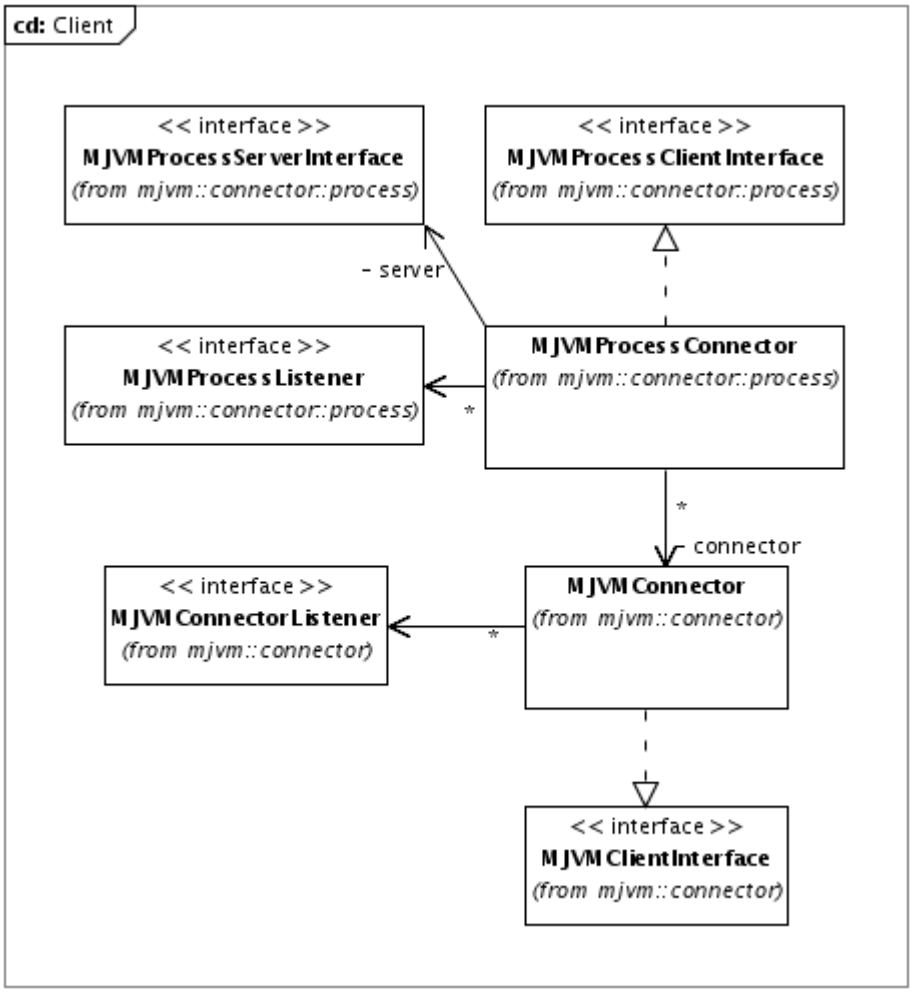


Abbildung 41: Klassenhierarchie MJVM-Client-Anwendung

Die Klasse *MJVMConnector* dient zum Starten einer MJVM. Außerdem kann man mit der Klasse *MJVMConnector* eine Verbindung zu einer laufenden MJVM herstellen und Java-Anwendungen in der MJVM starten. Die Klasse *MJVMProcessConnector* dient zum Steuern einer einzelnen Java-Anwendung in einer MJVM. An beide Klassen können Listener-Objekt angemeldet werden. Die Listener-Objekte werden bei Ereignissen wie: „Verbindung zur MJVM wurde getrennt“, „Java-Anwendung wurde gestartet“, „Java-Anwendung wurde beendet“ usw. benachrichtigt.

5.2.4.4.1 Eine MJVM starten

Wie bereits oben erwähnt, wurden einige Klassen der Java-Runtime-Environment (JRE) speziell an die MJVM angepasst. Das heißt, eine MJVM-Anwendung muss, statt mit den JRE-Klassen, mit den angepassten Klassen gestartet werden. Um eine Java-Anwendung mit anderen Klassen statt den JRE-Klassen zu starten, gibt es drei Möglichkeiten:

- Die angepassten JRE-Klassen in das Verzeichnis kopieren, in dem sich die Originalen Class-Dateien befinden. Dieses Verfahren hat den großen Nachteil, dass dadurch alle Java-Programme mit den MJVM-JRE-Klassen starten.
- Die MJVM-Anwendung mit dem Kommandozeilenparameter „-Xbootclasspath/p:./rt“ aufrufen. „./rt“ ist das Verzeichnis, welches die angepassten MJVM-JRE-Klassen enthält. Bis vor kurzem (bevor Java als Open-Source veröffentlicht wurde) hatte SUN, der Entwickler von Java, dieses Verfahren allerdings verboten.
- Bei der dritten Möglichkeit (die ich auch verwendet habe) werden die Klassen-Definitionen (Inhalt der Class-Dateien) der JRE-Klassen zur Laufzeit durch die Klassen-Definitionen der angepassten MJVM-JRE-Klassen ersetzt. Um zur Laufzeit Klassen-Definitionen zu ersetzten, kann man das JDI, das Java-Debug-Interface, verwenden.

Nachdem eine MJVM gestartet wurde, werden mit der Methode *patchClass* der Klasse *MJVMConnector* die System-Klassen, für die es spezielle MJVM-Versionen gibt, durch diese mit Hilfe von JDI überschrieben:

```
private void patchClass(File file, String name)
     throws MJVMException
{
     try{
          byte[] classData = MJVMClassLoader.loadClassData(file,
               name);
          List classList = mjvm.classesByName(name);
          if(classList.size() == 0)
               throw new RuntimeException("Klasse nicht gefunden");
          ReferenceType rt = (ReferenceType)classList.get(0);
          byte[] bytes = MJVMClassLoader.loadClassData(file, name);
          Map map = new HashMap();
          map.put(rt,bytes);
          mjvm.redefineClasses(map);
     }catch(Exception e){
          throw new MJVMException("Error during VM Patch");
     }
}
```

6 Test

6.1 Einleitung

Der Test des entwickelten Programms ist ein wichtiger Bestandteil bei der Software-Entwicklung. Die folgenden Abschnitte beschreiben die durchgeführten Tests und ihre Ergebnisse.

6.2 Systemtest

Ziel des Systemtests ist das Testen des Online-Compilers unter realistischen Bedingungen aus Sicht des Anwenders. Der Systemtest ist also der Test des Systems unter den gleichen Bedingungen, die später auch beim Einsatz des Online-Compilers herrschen. Im Systemtest wird validiert, ob das Gesamtsystem den Anforderungen aus Anwendersicht gerecht wird. Die Testfälle werden ausschließlich über die Anwenderschnittstelle, also einem Web-Browser, durchgeführt. Das heißt, es können auch nur Komponenten getestet werden, die über das Webinterface des Online-Compilers erreichbar sind. Deshalb wurde noch ein weiterer Test, der Server- und Lasttest, der zum Validieren der Funktionalität des Servers (das Online-Compiler-Servlet) dient, durchgeführt.

Dieses Kapitel ist aufgeteilt in:

- **Testplan:** Beschreibung der Voraussetzung und Umgebung des Systemtests sowie allgemeine Angabe der Beschreibung der Testfälle
- **Testspezifikation:** Beschreibung der einzelnen Testfälle
- **Testbericht:** Ergebnis des Systemtests

6.2.1 Testplan

Als **Testumgebung** wurde für den Client (mit dem das Webinterface des Online-Compilers aufgerufen werden soll) ein Linux-Computer mit einem Firefox 2.0 Browser gewählt. Der gesamte Test wurde nur mit diesem Browser durchgeführt. Andere Browser wurden später in einem speziellen Browsertest validiert (siehe nächsten Abschnitt). Als Server wurden der Computer „javock“ (Linux) und ein zweiter Rechner (ebenfalls Linux) verwendet. Der Test wurde also zweimal durchgeführt. Für die Benutzer-Authentifizierung wurde beim „javock“ der Ilias-Server der Fachhochschule Trier, http://ilias.fh-trier.de, verwendet. Für den anderen Rechner wurde ein lokaler Ilias-Server installiert.

Voraussetzung für den gesamten Test ist, dass alle Systeme (Client und Server) laufen und eine Verbindung zwischen beiden möglich ist.

Bei der **Beschreibung** der Testfälle wurde auf Details wie Vorbedingung, Nachbedingung usw. verzichtet, da diese sich aus der Beschreibung der Testfälle ergeben. Außerdem wurden mehrere zu testende Funktionen aus Gründen der Übersichtlichkeit zu einem Testfall zusammengefasst.

6.2.2 Testspezifikation

6.2.2.1 Testfall 1

Test einiger Funktionen des Online-Compilers. Insbesondere die Funktionen, die für Java-Programme benötigt werden. Dazu werden eine Reihe von Schritten in einem Web-Browser, in dem der Online-Compiler aufgerufen wurde, durchgeführt.

Schritte im Testfall 1:

1. Neues Verzeichnis *test* erstellen
2. Neue Java-Datei *test.java* im Verzeichnis *test* erstellen und Datei *test.java* öffnen, Inhalt der Datei:

```
import utilities.*;

public class test {
     public static void main(String[] args){
          System.out.println("Bitte Text eingeben:");
          TastaturEingabe.readString("");
     }
}
```

3. Datei *test.java* compilieren (automatisches Öffnen der „Probleme“-Registerkarte), kein Fehler vom Compiler erwartet, Ausgabe der Statusmeldung „Compilierung erfolgreich“
4. Datei *test.java* schließen
5. In der Statusmeldung „Compilierung erfolgreich“ auf „test.java“ klicken, die Datei *test.java* sollte geöffnet werden
6. Fehlermeldungen löschen, Statusmeldung „Compilierung erfolgreich“ sollte entfernt werden
7. Datei *test.java* ausführen (automatisch Öffnen der „Konsole“-Registerkarte), Ausgabe vom Prozess: „Bitte Text eingeben:“
8. Im Eingabefeld „Test“ eingeben, Ausgabe vom Prozess: „Test“
9. Prozess erneut starten und Ausführung abbrechen („Prozess abbrechen“ klicken)
10. Ausgabe des Prozesses löschen („Ausgabe löschen“ klicken)
11. Registerkarte „Probleme“ öffnen, eine Fehlermeldung „Prozess durch Benutzer abgebrochen“ sollte angezeigt werden
12. Datei bearbeiten und einen Fehler einfügen
13. Datei compilieren, in der „Probleme“-Registerkarte sollte eine Compiler-Fehlermeldung angezeigt werden
14. Datei erneut bearbeiten und Fehler entfernen

15. Datei *test.java* erneut ausführen und auf Timeout warten (2 Minuten)
16. In der Registerkarte „Probleme“ sollte eine entsprechende Fehlermeldung angezeigt werden
17. Die Kommandozeilen-Argumente der Datei *test.java* öffnen
18. In dem Eingabefeld den Text „a b c“ eingeben und die Argumente speichern
19. Datei im Editor öffnen, Dateiinhalt ersetzen durch:

```
public class test {
     public static void main(String[] args){
          for(int i = 0; i < args.length; i++)
               System.out.println(args[i]);
     }
}
```

20. Das Programm „test.java“ erneut ausführen, das Programm sollte „a b c“ ausgeben
21. Targets der Datei *test.java* öffnen, im Dialogfenster sollte nur die Datei *test.java* als Target eingetragen sein
22. Bei dem Target „test.java“ auf „linken“ klicken und Targets speichern, Fehlermeldung erwartet
23. Dialog schließen (abbrechen)
24. Inhalt der Datei *test.java* ersetzen durch:

```
package test;

public class test {
     public static void main(String[] args){
          for(int i = 0; i < args.length; i++)
               System.out.println(args[i]);
     }
}
```

25. Datei erneut ausführen, kein Fehler erwartet
26. In der ersten Zeile des Programms das Wort „test“ durch „test2“ ersetzen und Programm „test.java“ erneut ausführen, Fehlermeldung (Ausgabe des Programms) erwartet

6.2.2.2 Testfall 2

Test der Dokumentation und der Online-Hilfe sowie der verschiedenen Sprachen. Dazu werden wie beim ersten Testfall eine Reihe von Schritten in einem Web-Browser, in dem der Online-Compiler aufgerufen wurde, durchgeführt.

Schritte im Testfall 2:

1. Im Menü „DOKUMENTATION“ auf „Durchsuchen“ klicken. Der Suchen-Dialog sollte geöffnet werden.
2. „System“ in das Suchfeld eingeben und auf „Suchen“ klicken
3. Popup-Fenster mit dem Suchergebnis (Suche in Java-Dokumentation) sollte erscheinen
4. Suche mit Lisp, C, C# und C++ wiederholen
5. Im Menü „HILFE“ auf „Hilfe öffnen“ klicken, ein Popup-Fenster mit der Online-Hilfe des Online-Compilers sollte geöffnet werden
6. Im Menü „HILFE“ auf „Informationen über den Online-Compiler“ klicken, im selben Popup-Fenster, indem vorher die Online-Hilfe geöffnet wurde, sollte eine Informations-Seite geöffnet werden
7. Am unteren Fensterrand auf „english“ klicken, die Texte in der Benutzerschnittstelle sollten auf englisch angezeigt werden
8. Am unteren Fensterrand auf „中文“ klicken, die Texte in der Benutzerschnittstelle sollten auf chinesisch angezeigt werden
9. Am unteren Fensterrand auf „deutsch“ klicken, die Texte in der Benutzerschnittstelle sollten auf deutsch angezeigt werden

6.2.2.3 Testfall 3

Der dritte Testfall ist nahezu identisch mit dem ersten Testfall, der Test wird allerdings mit C++ statt Java durchgeführt. Dazu werden wie bei den vorherigen Testfällen eine Reihe von Schritten in einem Web-Browser, in dem der Online-Compiler aufgerufen wurde, durchgeführt.

Schritte im Testfall 3:

1. Neue C++-Datei *test.cpp* erstellen und Datei *test.cpp* öffnen, Inhalt der Datei:

```
#include <iostream>

int main()
{
	int i = 0;
	std::cout << "Eingabe:" << std::endl;
	std::cin >> i;
	std::cout << "Ausgabe:" << i << std::endl;
}
```

3. Datei *test.cpp* compilieren (automatisches Öffnen der „Probleme“-Registerkarte), kein Fehler vom Compiler erwartet, Ausgabe der Statusmeldungen „Compilierung erfolgreich“ und „Programm wurde erstellt“
4. Datei *test.cpp* schließen
5. In der Statusmeldung „Compilierung erfolgreich“ auf „test.cpp“ klicken, die Datei *test.cpp* sollte geöffnet werden
6. Fehlermeldungen löschen, Statusmeldungen sollten entfernt werden
7. Datei *test.cpp* ausführen (automatisches Öffnen der „Konsole“-Registerkarte), Ausgabe vom Prozess: „Eingabe:“
8. Im Eingabefeld „1“ eingeben, Ausgabe vom Prozess: „Ausgabe:1“
9. Prozess erneut starten und Ausführung abbrechen („Prozess abbrechen“ klicken)
10. Ausgabe des Prozesses löschen („Ausgabe löschen“ klicken)
11. Registerkarte „Probleme“ öffnen, eine Fehlermeldung „Prozess durch Benutzer abgebrochen“ sollte angezeigt werden
12. Datei bearbeiten und einen Fehler einfügen
13. Datei compilieren, in der „Probleme“-Registerkarte sollte eine Compiler-Fehlermeldung angezeigt werden
14. Datei erneut bearbeiten und Fehler entfernen
15. Datei *test.cpp* erneut ausführen und auf Timeout warten (2 Minuten)
16. In der Registerkarte „Probleme“ sollte eine entsprechende Fehlermeldung angezeigt werden
17. Die Kommandozeilen-Argumente der Datei *test.cpp* öffnen

18. In dem Eingabefeld den Text „a b c“ eingeben und die Argumente speichern
19. Datei im Editor öffnen, Dateiinhalt ersetzen durch:

```
#include <iostream>

int main(int argc, char* argv[])
{
     for(int i = 1; i < argc; i++)
          std::cout << argv[i] << " ";
}
```

20. Das Programm „test.cpp“ erneut ausführen, das Programm sollte „a b c“ ausgeben
21. Targets der Datei *test.cpp* öffnen, im Dialogfenster sollte nur die Datei *test.cpp* als Target eingetragen sein
22. Bei dem Target „test.cpp“ auf „linken“ klicken (linken der Datei deaktivieren) und Targets speichern
23. Datei *test.cpp* compilieren, Starten des Programmes (Schalter) sollte deaktiviert sein da Linker nicht ausgeführt wurde
24. Header-Datei *func.h* erstellen und Datei öffnen
25. Inhalt der Datei *func.h*:

```
void test();
```

26. C++-Datei *func.cpp* erstellen und Datei öffnen
27. Inhalt der Datei *func.cpp*:

```
#include <iostream>

void test()
{
     std::cout << "Test";
}
```

28. Inhalt der Datei *test.cpp* ändern:

```
#include <iostream>
#include "func.h"

int main(int argc, char* argv[])
{
     test();
}
```

29. Die Datei *test.cpp* compilieren, eine Fehlermeldung „Linken fehlgeschlagen“ sollte angezeigt werden
30. Zu Datei *test.cpp* das Target „func.cpp“ mit der Option „linken“ hinzufügen und die

Datei `test.cpp` neu Compilieren, keine Fehlermeldung erwartet

31. Programm „test.cpp“ ausführen, Ausgabe des Programmes sollte „Test“ sein

6.2.2.4 Testfall 4

Beim vierten Testfall werden alle Schritte des dritten Testfalls mit C statt C++ wiederholt.

6.2.2.5 Testfall 5

Im fünften Testfall wird geprüft, ob die Sicherheitsvorkehrungen durch AppArmor und den Java-Security-Manager für den geforderten Schutz sorgen. Dazu werden einige Java- und C++-Programme ausgeführt.

Test-Programme:

```
public class SecurityTest1 {
        public static void main(String[] args){
                try{
                        Runtime.getRuntime().exec("ps");
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

Das Programm *SecurityTest1* sollte mit einer Security-Exception abgebrochen werden.

```
public class SecurityTest2 {
     public static void main(String[] args){
          try{
               PrintWriter output = new PrintWriter(new
                         FileWriter("/test.txt",true));
               output.print("Inhalt");
               output.close();
          }catch(Exception e){
               e.printStackTrace();
          }
     }
}
```

Das Programm *SecurityTest2* sollte mit einer Security-Exception abgebrochen werden.

```c
#include <stdio.h>

int main()
{
     int i;
     FILE *fd;
     fd = fopen("/test.txt", "w");
     if (fd == NULL){
          fprintf(stderr, "fopen failed for %s\n", "/test.txt");
          return 1;
     }
     fprintf(fd, "scribbled on file %s\n", "/test.txt");
     fclose(fd);
}
```

Die Ausgabe des C-Programmes sollte „fopen failed for /test.txt“ sein.

```c
#include<stdio.h>

void exec_cmd (char *buf);
#define MAX_ARGS 100

int main(int argc,char **argv)
{
     char buf[256];
     printf("enter cmd:");
     if(gets(buf)!= NULL)
      exec_cmd(buf);
}

void exec_cmd(char *buf){
     char *argv[MAX_ARGS];
     int j = 0;
     argv[j++] = strtok(buf," ");
     while(j<MAX_ARGS&&(argv[j++]=strtok(NULL," "))!=NULL);
     execvp(argv[0],argv);
     _exit(1);
}
```

Das Programm erwartet den Namen eines Befehls (z.B. „ps“). Die Ausgabe des Befehls (z.B. die Ausgabe von „ps“) sollte nicht in der Ausgabe des C-Programms enthalten sein, da AppArmor das Ausführen des Befehls blockieren muss.

6.2.2.6 Testfall 6

Im sechsten Testfall sollen alle Funktionen des Debugger getestet werden.

Schritte im Testfall 6:

1. Datei *test1.java* erstellen, Datei öffnen und den folgenden Inhalt einfügen:

```
public class test1 {
	public static void main(String[] args){
		test2.test(1);
		test2.test(2);
		test2.test(3);
	}
}
```

2. Datei *test2.java* erstellen, Datei öffnen und den folgenden Inhalt einfügen:

```
public class test2 {
	public static void test(int a){
		int b = a + 1;
		for(int i = 0; i < 10; i++)
			b++;
		System.out.println(b);
	}
}
```

3. In der vierten Zeile der Datei *test2.java* einen Debugger-Haltepunkt erstellen
4. Datei *test1.java* mit dem Debugger starten (automatisches Öffnen der „Fehlersuche“-Registerkarte)
5. In der Registerkarte „Fehlersuche“ sollte, nach einiger Zeit, die Position des Debuggers (4. Zeile in Datei *test2.java*) angezeigt werden, die Datei *test2.java* sollte geöffnet werden und die vierte Zeile markiert werden
6. eine neue Debugger-Variable „b“ erstellen, der Wert „2“ sollte angezeigt werden
7. Step-Over ausführen, die fünfte Zeile sollte markiert sein
8. Step-Out ausführen, Datei *test1.java* sollte mit Zeile 4 als aktiver Zeile geöffnet werden
9. Debugger-Haltepunkt in Zeile 5 erstellen
10. Debugger fortsetzen (Continue bzw. Resume)
11. Debugger fortsetzen (Continue bzw. Resume)
12. Programm sollte in der fünften Zeile der Datei *test1.java* stehen bleiben

6.2.2.7 Testfall 7

Testfall zum Testen der Online-Compiler-Dateiverwaltung und der Benutzer An- und Abmeldung. Dazu werden eine Reihe von Schritten in einem Web-Browser, in dem der Online-Compiler aufgerufen wurde, durchgeführt.

Schritte im Testfall 7:

1. Neues Verzeichnis `test` erstellen
2. Neue Datei `test.java` im Verzeichnis `test` erstellen
3. Datei `test.java` öffnen
4. Verzeichnis `test` löschen, die Datei `test.java` sollte automatisch geschlossen werden
5. Datei `test.java` erstellen
6. Datei `test.java` erstellen, Fehlermeldung erwartet
7. Datei `test2.java` erstellen
8. Datei `test2.java` in `test.java` und `test.class` umbenennen, jeweils eine Fehlermeldung erwartet
9. ein Beispiel aus dem Beispiel-Ordner öffnen, Dateien des Beispiels sollten im Dateibaum angezeigt werden
10. Verzeichnis mit Dateien erstellen
11. Verzeichnis als JAR- und als XML-Datei herunterladen
12. Verzeichnis löschen
13. heruntergeladene Dateien hochladen, alle Dateien sollten wiederhergestellt werden
14. Benutzer abmelden, die Dateien im Dateibaum sollten daraufhin geschlossen werden und ein Dialog zum Eingeben von Benutzername und Passwort angezeigt werden
15. mit einem gültigen Ilias-Benutzernamen und falschem Passwort anmelden, Fehlermeldung erwartet
16. mit einem gültigen Ilias-Benutzernamen mit Passwort anmelden

6.2.3 Testbericht

6.2.3.1 gefundene Fehler

Während des Tests wurden einige Fehler gefunden. Nach Korrektur dieser, wurde der Test wiederholt, ohne dass ein weiterer Fehler aufgetreten ist. Die gefundenen Fehler waren:

Fehler im Testfall 1:

- Bei Kommandozeilen-Argumenten wurden die Leerzeichen zwischen den Argumenten auch als einzelne Argumente erkannt. Der Fehler lag in der Methode *setArgs* der Klasse *onlinecompiler.languages.files.LaunchableFile* in der die Kommandozeilen-Argumente analysiert werden.
- Beim Wechseln von Registerkarten in der Benutzerschnittstelle wurde diese nicht immer richtig dargestellt. Die Ursache von diesem Fehler ist ein Fehler im Firefox-Browser bei der Darstellung von HTML-IFrame-Elementen. Um diesen Problem zu umgehen werden im Firefox-Browser Registerkarten komplett neu erzeugt sobald auf einen Tab geklickt wurde. Im Internet-Explorer, bei dem es dieses Problem nicht gibt, werden die Registerkarten hingegen versteckt, falls sie nicht angezeigt werden sollen, beziehungsweise angezeigt, falls sie aktiv sind.

Fehler im Testfall 2:

- In der Benutzerschnittstelle wurden Sonderzeichen falsch dargestellt. Ursache war die Angabe eines falschen Zeichensatzes beim Senden von HTML- und JavaScript-Dateien an Web-Clients durch das Online-Compiler-Servlet. Der Zeichensatz (UTF-8) von HTML- und JavaScript-Dateien wird in der Methode *answerDocumentRequest* der Klasse *onlinecompiler.servlet.OCServlet* angegeben:

```
if(isHtmlRequest(request))
     response.setContentType("text/html;charset=utf-8");
else if(isJavaScriptRequest(request))
     response.setContentType("text/javascript;charset=utf-8");
```

  Neben der Angabe des Zeichensatzes für HTML- und JavaScript-Dateien müssen diese außerdem als UTF-8-Dateien gespeichert werden.

Fehler im Testfall 7:

- Die Benutzer-Authentifizierung mit dem Ilias-Server der Fachhochschule Trier (Test mit „javock“) ist fehlgeschlagen. Bei dem zweiten Server (lokaler Ilias-Server) dagegen trat der Fehler nicht auf. Die Anmeldung an den FH-Trier Ilias-Server kann aufgrund eines Fehlers in Ilias nicht funktionieren. Da der FH-Trier Ilias-Server auf einen weiteren Server (LDAP) zur Benutzer-Authentifizierung zugreift, ist eine Benutzer-Authentifizierung über die SOAP-Schnittstelle des Ilias-Servers zur Zeit nicht möglich.

6.3 Browsertest

Ziel des Browsertests ist es, den Online-Compiler beziehungsweise die Benutzerschnittstelle mit verschiedenen Web-Browsern zu testen. Im Browsertest wurden nicht nur die beiden Web-Browser, die laut Anforderung unterstützt werden müssen, getestet, sondern auch alle anderen gängigen Web-Browser.

Im Browsertest werden beide Versionen der Benutzerschnittstelle (die Ilias-Version und die „normale“-Version) getestet.

6.3.1 Firefox

Im aktuellen Firefox-Browser (Version 2.0) *[Mozilla 2]* können alle Funktionen des Online-Compilers ohne Probleme aufgerufen werden. Während des Tests wurde kein Fehler gefunden.

6.3.2 Internet-Explorer

Mit dem Internet-Explorer gab es am meisten Probleme während der Entwicklung des Online-Compilers. Mit der aktuellen Version 7.0 *[Microsoft 1]* gab und gibt es eine Reihe von Fehler bei der Anzeige des Online-Compilers:

- Debugger-Haltepunkte werden einige Pixel zu groß angezeigt.
- Unter den Menüpunkten ist eine 2 Pixel große Linie sichtbar.
- Nach dem Öffnen einer Datei im Quelltext-Editor wurde die dritte Textzeile in der Farbe Weiß dargestellt. Erst wenn man den Mauszeiger über den Editor bewegt hat, wurde die Zeile korrekt dargestellt. Dieses Problem konnte behoben werden, indem direkt nach dem Öffnen einer Datei im Editor in der HTML-Seite ein DIV-Element erzeugt und sofort wieder gelöscht wird. Der Internet-Explorer wird dadurch veranlasst, die Elemente der HTML-Seite noch einmal zu zeichnen wodurch der Darstellungsfehler in der dritten Zeile behoben wird.
- Wenn man den MIDAS-Editor mit dem JavaScript-Befehl *designMode* aktiviert, wird im Editor der Online-Compiler selbst (HTML-Seite „index.html“) geladen. Obwohl *designMode* der offizielle Weg zum aktivieren des MIDAS-Editors im Internet-Explorer ist, beseitigt man den Fehler dadurch, das man im Internet-Explorer statt des *designMode*-Befehles *contentEditable* verwendet.
- Ein weiterer Fehler betrifft das Hochladen von Dateien zum Online-Compiler-Servlet. Wenn man eine Datei hochlädt, kommt im Servlet zwar eine Datei an, aber leider wird kein Inhalt vom Internet-Explorer mitgeschickt. Dieses Problem tritt nur in der Version 7.0 des Internet-Explorers auf, sobald man ein Upload-Formular mit JavaScript erzeugt. Mit einer statischen HTML-Datei, in der man eine Datei zum Hochladen auswählen kann, tritt dieser Fehler nicht auf. Um das Hochladen von Dateien auch im Internet-Explorer zu ermöglichen, habe ich deshalb die HTML-Datei `upload.html` hinzugefügt, welche auch schon im Java-Online-Compiler verwendet wurde, um Dateien zum Hochladen auszuwählen.

6.3.3 Opera

Mit dem Opera-Browser gibt es einige Probleme bei Anzeigen der Benutzerschnittstelle. In der Linux-Version des Opera-Browsers (Version 9.0, *[Opera 1]*) wird zum Beispiel nur die Anzeige „Lade Online-Compiler“ angezeigt während der Rest der Benutzerschnittstelle nicht gezeigt wird. Auf anderen Betriebssystemen gibt es mit der aktuellen Version 9.0 ebenfalls Probleme:

- Tabs (Überschriften der Registerkarten) werden nicht korrekt angezeigt
- Größenänderungen von Komponenten der Benutzerschnittstelle werden nicht korrekt ausgeführt
- der MIDAS-Editor wird nicht vollständig von Opera unterstützt

6.3.4 Safari

Im Safari-Browser von Apple *[Apple 1]* gibt es ähnliche Probleme wie im Opera-Browser. Die meisten Probleme betreffen, wie auch im Opera-Browser, die Unterstützung des Quelltext-Editors. Der MIDAS-Editor wird allerdings von Safari unterstützt. Dass heißt, mit einigen speziellen Änderungen sollte der Online-Compiler auch für den Safari-Browser kompatibel gemacht werden können.

6.3.5 Konqueror

Der Online-Compiler kann nicht mit der aktuellen Version (3.x) des Konqueror-Browsers *[KDE 1]* aufgerufen werden, da dieser den MIDAS-Editor, also den Quelltexteditor, nicht unterstützt. Die im Laufe des Jahres erscheinende Version 4 wird, da sie auf Firefox und Safari basieren wird, mit großer Wahrscheinlichkeit den Online-Compiler ausführen können.

6.3.6 Fazit

Bei beiden Browsern, die vom Online-Compiler unterstützt werden sollen, werden alle Funktionen der Benutzerschnittstelle korrekt ausgeführt und angezeigt. Die anderen Browser haben zwar alle Probleme mit einigen Funktionen des Online-Compilers (vor allem des Quelltext-Editors) aber grundsätzlich lässt sich sagen, dass durch einige Anpassungen der Online-Compiler für jeden Browser kompatibel gemacht werden könnte.

6.4 Last- und Servertest

Ziel des Last- und Servertests ist es, Funktionen zu testen, die im Systemtest, Aufgrund der Einschränkungen des Webinterface des Online-Compilers, nicht getestet werden konnten. Dazu zählt unter anderem ein ausführlicher Lasttest des Systems.

Der Lasttest ist ein wichtiger Bestandteil des gesamten Tests. Dabei soll geklärt werden, ob sich der Online-Compiler eignet, von mehreren Anwendern parallel verwendet zu werden. Dies ist eine unverzichtbare Anforderung an eine Client-Server-Anwendung. Schließlich macht der Einsatz des Online-Compilers im Rahmen von E-Learning nur dann Sinn, wenn der Online-Compiler auch von mehreren Anwendern gleichzeitig genutzt werden kann.

Dieses Kapitel ist aufgeteilt in:

- **Testplan:** Beschreibung der Voraussetzung und Umgebung des Last- und Servertests
- **Testspezifikation:** Beschreibung der einzelnen Testfälle
- **Testbericht:** Ergebnis des Last- und Servertests

6.4.1 Testplan

Um den Server- und Lasttest durchzuführen, wurde eine **Testklasse** erstellt. Dies hat zum einen den Vorteil, das der Test einfach und schnell wiederholt werden kann und zum anderen, dass ein Testprotokoll leicht erstellt werden kann. Außerdem erfordert der Test Funktionen, die nur mit Hilfe eines Programmes realisiert werden können. So ist ein Lasttest, bei dem sehr viele Anwender schnell hintereinander eine Reihe von Anfragen stellen, nur rechnergestützt möglich.

Das Paket *onlinecompiler.test* enthält die Klassen, die für den Test benötigt werden. Die Testklasse *TestServer* enthält das Programm, welches die verschiedenen Testfälle ausführt. Jeder Testfall ist in einer eigenen Klasse implementiert.

Als **Testumgebung** wurde neben dem Server „javock“, auf dem das OC-Servlet ausgeführt wird, ein zweiter Rechner (Mac OS X) zum Ausführen der Testklassen verwendet.

Voraussetzung für den gesamten Test ist, dass alle Systeme laufen und eine Verbindung zwischen Server und Client möglich ist.

Um zu überprüfen, ob sich die Performance des Online-Compilers auf einem schnelleren System signifikant verbessern lässt, wurde der Test ebenfalls mit einem zweiten Server-Computer durchgeführt (Linux).

Als **Testprotokoll** wurde das komplette Kommunikationsprotokoll zwischen Client und Server aufgezeichnet. Zudem wurde während des Lasttests die Auslastung des Server-Systems beobachtet.

6.4.2 Testspezifikation

6.4.2.1 Testfall 1

Test, ob das Servlet bei der maximalen Anzahl an Benutzern keine weiteren mehr zulässt (Grenzfalltest).

Klasse: *TestfallMaxUser*

Vorbedingung: Das Servlet muss laufen. Kein Benutzer darf angemeldet sein.

Nachbedingung: Das Servlet muss laufen und das Verzeichnis für die Gast-User (`tempusers`) muss leer sein

6.4.2.2 Testfälle 2, 3 und 4

Testen, wie sich das Servlet bei vielen Benutzern, die gleichzeitig C-, C++ und Java-Programme compilieren, verhält. Dazu werden mit mehreren Threads Anfragen zum Compilieren von Programmen parallel ausgeführt.

Klassen: *TestfallLastCBuild*, *TestfallLastCPPBuild* und *TestfallLastJavaBuild*

Vorbedingung: Das Servlet muss laufen

Nachbedingung: keine.

Randbedingung: während des Tests sollte der Online-Compiler normal über einem Browser verwendet werden können

6.4.2.3 Testfälle 5, 6 und 7

Testen, wie sich das Servlet bei vielen Benutzern, die gleichzeitig C-, C++ und Java-Programme ausführen, verhält. Dazu werden mit mehreren Threads Anfragen zum Ausführen von Programmen parallel ausgeführt.

Klassen: *TestfallLastCLaunch*, *TestfallLastCPPLaunch* und *TestfallLastJavaLaunch*

Vorbedingung: Das Servlet muss laufen

Nachbedingung: keine.

Randbedingung: während des Tests sollte der Online-Compiler normal über einem Browser verwendet werden können

6.4.2.4 Testfall 8 (MJVM)

Ziel dieses Tests ist es, zu überprüfen, ob Java-Programme von einer MJVM (Multi-Java-VM) ausgeführt werden können. Die eigentliche Aufgabe des achten Testfalls ist es aber, das Verhalten der MJVM bei vielen Java-Anwendungen, die ausgeführt werden, zu überprüfen. Dadurch soll ermittelt werden, ob durch das parallele Ausführen von Java-Anwendungen innerhalb einer Java-VM (der MJVM) die Ausführungsgeschwindigkeit verbessert werden kann.

Für den Testfall wird das Programm *DemoFrontend* des Paketes *mjvm.connector.demo* verwendet. Dieses Programm wurde bereits im Kapitel „Implementierung“ vorgestellt.

Das Programm *DemoFrontend* führt 50 mal ein Java-Programm

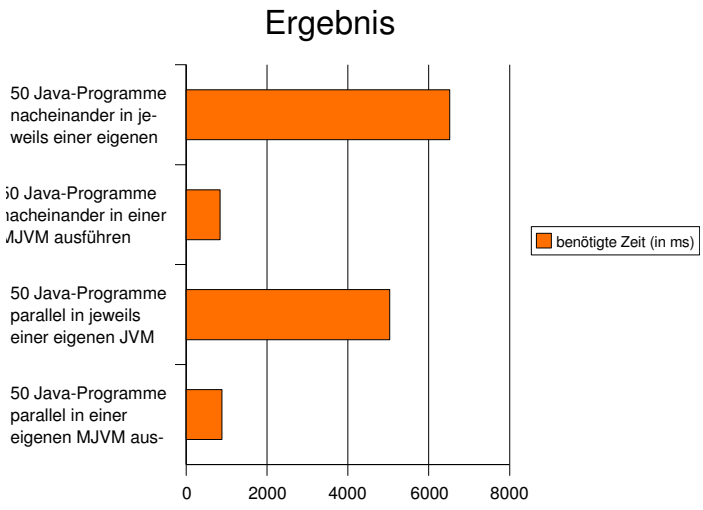
- nacheinander in jeweils einer eigenen Java-VM
- nacheinander in einer MJVM
- parallel in jeweils einer eigenen Java-VM
- parallel in einer einzigen MJVM

aus. Dabei wird jeweils die benötigte Zeit gemessen.

Aufgerufen wird das Programm *DemoFrontend* mit den folgenden Befehlen:

```
java -cp .
     -Xbootclasspath/p:.
     -Xbootclasspath/a:/usr/lib/java/lib/tools.jar:.
     mjvm.server.MJVMVirtualMachine
java mjvm.connector.demo.DemoFrontend 50 ./rt HalloWelt
```

Laufzeit von 50 Java-Programmen (Testergebnis):



6.4.3 Testbericht

6.4.3.1 gefundene Fehler

Während des Tests ist kein Fehler aufgetreten.

6.4.3.2 weitere Probleme

Sonstige Probleme, die während des Testlaufs aufgetreten sind:

Probleme in allen Testfällen des Lasttestes:

- Bei jedem Lasttestfall konnte die Randbedingung „der Online-Compiler muss über einem Browser während des Tests verwendet werden können“ nie erfüllt werden. Das Servlet bzw. der Computer „javock“, auf dem das Servlet lief, wurde durch die Tests so stark belastet, dass ein normales Arbeiten mit dem Servlet während der Tests vollkommen unmöglich war. Bei der Ausführung mit einem schnelleren Rechner (als Ersatz für den Server „javock“) wurden bessere Resultate erzielt. So war bei dem Test mit dem schnelleren Rechner ein Arbeiten mit dem Online-Compiler während des laufenden Tests möglich.

6.5 Test-Ergebnis

Das Ergebnis des Lasttestes ähnelt stark dem Ergebnissen der Vorgängerversion des Online-Compilers, dem Java-Online-Compiler. So konnte der Rechner „javock“ wie schon beim Java-Online-Compiler keinen der Lasttestfälle bestehen. Allerdings entsprach die Auslastung des „javock“-Rechners während des Tests ungefähr der Auslastung beim Test des Java-Online-Compilers. Dass heißt, obwohl der Online-Compiler eine bei weitem komplexere Anwendung als der Java-Online-Compiler ist, ist die Auslastung ungefähr gleich geblieben.

Der Lasttest hat aber auch gezeigt, das die Geschwindigkeit durch den Einsatz der MJVM (Multi-Task-Java-VM) stark erhöht werden kann. Die Ergebnisse der Lasttests der MJVM werden in der folgenden Tabelle aufgelistet.

Test	***benötigte Zeit (in ms)***
50 Java-Programme nacheinander in jeweils einer eigenen JVM ausführen	6520
50 Java-Programme nacheinander in einer MJVM ausführen	840
50 Java-Programme parallel in jeweils einer eigenen JVM ausführen	5034
50 Java-Programme parallel in einer eigenen MJVM ausführen	885

Die MJVM ist also eine Möglichkeit, die Ausführung von Java-Programmen deutlich zu beschleunigen. Der Nachteil dieser Lösung ist aber, dass sie nur für Java-Programme genutzt werden kann. C- und C++-Programme, beziehungsweise deren Ausführung, können mit der MJVM nicht beschleunigt werden.

Eine weitere Lösung des Geschwindigkeits- und Lastproblems ist, wie sich auch schon beim Java-Online-Compiler gezeigt hat, der Einsatz einer besseren Hardware. So wurde auf dem Vergleichsrechner ein deutlich besseres Ergebnis als auf dem „javock“-Server erzielt.

7 Server-Sicherheit

In den vorherigen Kapiteln wurde bereits mehrfach auf Sicherheitsaspekte des Online-Compilers eingegangen. Die Sicherheit des Online-Compilers und des Rechners, auf dem der Online-Compiler ausgeführt wird (zum Beispiel der Server „javock“), hängt nicht alleine von dem Online-Compiler selbst ab. Der Online-Compiler wird immer im Zusammenhang mit anderen Programmen ausgeführt. Dazu zählen AppArmor, Tomcat und die Java-VM von SUN. Bei der Betrachtung der Sicherheit müssen diese Programme deshalb mit einbezogen werden. In diesem Kapitel wird auf die Sicherheitsprobleme und Anforderungen der einzelnen Programme, die vom Online-Compiler verwendet werden, eingegangen.

7.1 Tomcat

Der Tomcat-Webserver spielt eine wichtige Rolle im Zusammenhang mit der Sicherheit des Online-Compilers. So können zum Beispiel **Brute-Force-Angriffe** (dabei versucht ein Angreifer einen Server zum Stillstand zu bringen, in dem in einem kleinen Zeitraum möglichst viele Client-Anfragen gestellt werden) nur von dem Tomcat-Server abgefangen werden, da der Tomcat-Server das Programm ist, welches die Client-Anfragen entgegen nimmt und erst danach an das entsprechende Servlet weiterleitet. Vom Bundesamt für Sicherheit in der Informationstechnik (BSI) wurde eine ausführliche Untersuchung des Tomcat-Server durchgeführt *[BSI Tomcat 1]*. Dabei wurde eine Liste mit Empfehlungen für einen sicheren Betrieb des Tomcat-Servers zusammengestellt. Zu den Empfehlungen des BSI zählen:

- Verwendung von SSL
- Installation des Tomcat-Servers mit einem anderen Benutzerkonto (nicht dem Root-Benutzer des Betriebssystems)
- Schutz des Tomcat-Servers durch eine LSM-Anwendung (zum Beispiel AppArmor)
- Einsatz einer Firewall

7.2 AppArmor

AppArmor schützt ein System vor schadhaften Programmen. So wird AppArmor sogar als Sicherheitsfunktion für den Apache-Webserver *[Apache 2]* verwendet. AppArmor setzt auf das sogenannte LSM (Linux Security Modul) auf. Das LSM ist eine Schnittstelle des Betriebssystem-Kerns (Kernel) von Linux, mit der bestimmte Funktionen für Programme explizit gesperrt werden können. So kann man zum Beispiel mit AppArmor über LSM das Schreiben von Dateien verbieten. AppArmor hat aber den Nachteil, dass nicht alle Funktionen des LSM verwendet werden. So ist es zum Beispiel mit AppArmor nicht möglich, die Betriebssystem-Funktion „fork“ zu sperren. Mit dem Befehl „fork“ kann ein Programm eine Kopie von sich selbst erstellen. Eine bestimmte Sorte von Programmen, die sogenannten **Fork-Bombs**, nutzt diese Funktion aus, um einen Rechner zum Stillstand zu bringen. Dazu versucht die Fork-Bomb unendlich viele Kopien von sich selbst zu erstellen. Das folgende C-Programm ist eine Fork-Bomb:

```
#include <unistd.h>

int main(void)
{
     while(1) fork();
     return 0;
}
```

Da man mit AppArmor den Befehl „fork“ nicht sperren kann, können Benutzer des Online-Compilers mit einer Fork-Bomb den Rechner, auf dem der Online-Compiler ausgeführt wird, zum Stillstand bringen. Die derzeit einzige Maßnahme, um eine Fork-Bomb dennoch zu verhindern, ist der Befehl „ulimit“. Mit diesem Befehl kann man die maximale Anzahl an Prozessen, die ein Benutzer des Betriebssystems ausführen darf, begrenzen.

7.3 Java-Security-Manager

Mit dem Java-Security-Manager können Java-Programme gut vom Rest des Systems abgeschirmt werden. Außerdem hat man von einem Java-Programm nicht direkten Zugriff auf Betriebssystem-Funktionen. Fork-Angriffe und sonstige Gefahren durch Betriebssystem-Funktionen sind damit nahezu ausgeschlossen.

7.4 Zusammenfassung Sicherheitsvorkehrungen

Für einen möglichst sicheren Betrieb des Online-Compilers sollten eine Reihe von Sicherheitsvorkehrungen getroffen werden:

- Regelmäßiges Update aller verwendeten Software-Komponenten. Dazu zählen auch die verwendeten Java-Bibliotheken des Online-Compilers.
- Regelmäßiges Auswerten der Log-Dateien des Tomcat-Servers.
- Installation und Betrieb des Tomcat-Servers unter einem speziellem Benutzerkonto. Der Tomcat-Server sollte nicht vom Root-Benutzer des Betriebssystems ausgeführt und installiert werden.
- Maximale Anzahl der Prozesse begrenzen (ulimit).
- Zugang zum Online-Compiler für Gäste sperren.

8 Zusammenfassung

Ziel dieser Arbeit war das Erstellen einer Online verfügbaren Entwicklungsumgebung für verschiedene Programmiersprachen. Dazu wurde ein Java-Servlet als „moderne“ Webanwendung mit dem Ajax-Konzept entwickelt. Als Grundlage der Entwicklung diente das Programm Java-Online-Compiler, welches ich im Rahmen einer Projektarbeit zusammen mit Lars Herwartz entwickelt habe *[HV 2006]*.

Das entwickelte Programm, der Online-Compiler, erfüllt alle Anforderungen, die im Kapitel „Anforderungen und Ziele“ angegeben wurden:

- Java, C und C++-Programme können entwickelt werden
- Dazu gehört das Compilieren von Java- und C- bzw. C++-Dateien, das Linken von C- und C++-Programmen sowie das Ausführen von Java und C- bzw. C++-Programmen mit oder ohne einen Debugger.
- Wie in anderen Entwicklungsumgebungen werden Ein- und Ausgabe von ausgeführten Programmen angezeigt.
- Mit den Debugger-Funktionen kann nach Fehlern in Programmen gesucht werden. Der Online-Compiler bietet dazu alle nötigen Funktionen: Hinzufügen von Haltepunkten, Anzeigen von Variablen-Werten, Step-Out, Step-Into und Step-Over.
- Benutzer des Ilias-Servers können sich mit ihrem Ilias-Benutzernamen und dem dazugehörigen Passwort anmelden
- Funktionen zur Integration des Online-Compilers in ein E-Learning-System (Ilias).

All diese Funktionen zusammen bilden eine vollständige Entwicklungsumgebung. Das Programm wurde einem ausführlichen System- und Lasttest unterzogen.

Zudem wurde mit der MJVM (Programm zum Ausführen mehrere Java-Programme in einer Java-VM) eine Idee aus der oben erwähnten Projektarbeit aufgegriffen. Durch Tests konnte bewiesen werden, dass mit dem Konzept die Geschwindigkeit von Java-Programmen deutlich erhöht werden kann.

8.1 Ausblick

Ob das Ziel, eine einfach zu bedienende Entwicklungsumgebung zu entwickeln, erreicht wurde, wird sich letztendlich erst beim Einsatz des Online-Compilers mit „richtigen“ Anwendern (zum Beispiel im Rahmen des Fernstudium-Angebotes der Fachhochschule Trier) zeigen.

Während der Entwicklung des Online-Compilers bin ich auf einige wünschenswerte Erweiterungen aufmerksam geworden, die umzusetzen das System verbessern würden:

- bessere Ilias-Integration:
  Dateien und Programm-Beispiele sollten direkt in Ilias integriert werden. Auf diese Weise spart man sich zum Beispiel das Übernehmen der Programm-Beispiele aus den Online-Kursen in den

Online-Compiler.

- bessere Integration in die Online-Kurse:
  Statt den Online-Compiler in einem Frame in den Online-Kursen darzustellen („Ilias-Version“), sollte der Online-Compiler direkt in die Kurs-Seiten integriert werden. So könnte man zum Beispiel bei Quelltexten (Programm-Beispiele) in den Seiten der Online-Kurse direkt den Quelltexteditor des Online-Compilers integrieren. Die Anwender könnten dann direkt, ohne in ein anderes Fenster zu wechseln, die Programm-Beispiele ausführen und bearbeiten.

Weitere Aufgaben, die noch ausgeführt werden sollten:

- C# und Lisp:
  Die Unterstützung für die Programmiersprachen C# und Lisp muss noch zu dem Online-Compiler hinzugefügt werden.

- Programm-Beispiele:
  Die Programm-Beispiele der verschiedenen Online-Kurse müssen in den Online-Compiler übernommen werden. Diese Aufgabe konnte im Rahmen dieser Abschlussarbeit nicht ausgeführt werden, da die Online-Kurse (zum Beispiel „Datenstrukturen + Algorithmen“) zum Zeitpunkt des Fertigstellens dieser Abschlussarbeit noch nicht vollständig fertiggestellt waren.

- Integration der MJVM in den Online-Compiler:
  Die MJVM muss zum Ausführen von Java-Programmen in den Online-Compiler integriert werden.

- Benutzerauthentifizierung mit dem Ilias-Server der Fachhochschule Trier.

- Unterstützung weiterer Webbrowser wie Safari, Opera und Konqueror.

9 Anhang

Dieser Anhang enthält:

- das **Inhaltsverzeichnis** der CD auf der sich die vollständige Abschlussarbeit befindet
- das **Administratorhandbuch** mit Informationen zur Installation und Administration des Online-Compiler-Servlets
- das **Benutzerhandbuch** für die Anwender des Online-Compilers

9.1 CD-Inhalt

Verzeichnis	*Inhalt*
`bin`	
`MultiTaskJavaVM`	das Java-Programm MJVM (Multi-Task-Java-VM)
`OnlineCompiler`	
`config`	Konfigurationsdateien des Online-Compilers
`oc`	der Online-Compiler (Servlet)
`oclibs`	Java-Bibliotheken des Online-Compilers
`doc`	
`api`	API-Dokumentation (Javadoc)
`Dokumentation`	Dokumentation (dieses Dokument)
`Bilder`	Bilder und Bildschirmfotos
`uml`	UML-Diagramme
`source`	
`MultiTaskJavaVM`	MJVM-Quelltextdateien
`OnlineCompiler`	Online-Compiler-Quelltextdateien
`tools`	
`SSL`	Programm zum Herunterladen eines SSL-Server-Zertifikates

9.2 Administratorhandbuch

Dieses Kapitel soll bei der Administration des vorhandenen Servers „javock“ bzw. eines anderen Servers, auf dem der Online-Compiler laufen soll, helfen.

9.2.1 Installation des Online-Compilers

Voraussetzungen für eine Installation des Online-Compilers sind:

- Linux-Betriebssystem, http://de.opensuse.org
- AppArmor (Version >= 2.0), http://de.opensuse.org/Apparmor
- Tomcat Webserver (Version 5.0.28), http://tomcat.apache.org

Die Installation des Online-Compilers auf einem System mit bereits installiertem Tomcat Webserver ist relativ einfach und in wenigen Schritten durchgeführt:

1. Das Unterverzeichnis `oc` aus dem Verzeichnis `bin/OnlineCompiler` muss in das Unterverzeichnis `webapps` des Tomcat-Servers kopiert werden. Das Verzeichnis `webapps` befindet sich in der Regel im gleichen Verzeichnis wie die Binaries des Tomcat-Servers.
2. Die Datei `web.xml` im Unterverzeichnis `webapps/oc/WEB-INF` in einem Editor öffnen.
3. In der Datei `web.xml` muss beim Eintrag „<param-name>root</param-name>“ bei „<param-value>“ der Name des Verzeichnisses angegeben werden, indem sich die Konfigurations- und Beispieldateien des Online-Compilers befinden sollen (zum Beispiel das Verzeichnis `/srv/oc`).
4. In dieses Verzeichnis sollte der komplette Inhalt des CD-Verzeichnisses `bin/OnlineCompiler/config` kopiert werden.
5. In der Datei `web.xml` muss beim Eintrag „<param-name>wwwroot</param-name>“ bei „<param-value>“ der Name des Verzeichnisses angegeben werden, indem sich die HTML- und JavaScript-Dateien des Online-Compilers befinden. Dieses Verzeichnis ist normalerweise das Verzeichnis `webapps` des Tomcat-Servers.
6. Der Online-Compiler benötigt einige Java-Bibliotheken (unter anderem zum Parsen von XML-Dateien). Diese Bibliotheken müssen in das Verzeichnis `common/endorsed` des Tomcat-Servers kopiert werden. Die Java-Bibliotheken befinden sich im Unterverzeichnis `bin/OnlineCompiler/oclibs` auf der CD.
7. Zum Schluss muss noch der Tomcat-Server neu gestartet werden. Dazu kann man unter Linux die Befehle `/usr/share/tomcat5/bin/shutdown.sh` und `/usr/share/tomcat5/bin/startup.sh` verwenden.
8. Danach kann der Online-Compiler über die URL http://localhost:8080/oc/index.html aufgerufen werden.

Soll eine Benutzerauthentifizierung mit einem Ilias-Server erfolgen, müssen die URL und der Name des Ilias-Servers in der Konfigurationsdatei des Online-Compilers angegeben werden (siehe nächster Abschnitt). Falls die Verbindung zum Ilias-Server per HTTPS erfolgen soll, muss das SSL-Zertifikat des Ilias-Servers heruntergeladen werden. Damit der Online-Compiler Zugriff auf das SSL-Zertifikat des Ilias-Server erhält, kann man das Java-Programm *InstallCert* aus dem Verzeichnis `tools/SSL` auf der CD aufrufen. Diese Programm lädt ein SSL-Zertifikat von einem Server und speichert es, damit alle Java-Programme darauf zugreifen können.

9.2.2 Konfiguration des Online-Compilers

Das Konfigurations- und Datenverzeichnis des Online-Compilers wird in der Datei `web.xml` im Unterverzeichnis `WEB-INF` angegeben. Die Voreinstellung ist `/srv/oc`. Im Konfigurations- und Datenverzeichnis befinden sich alle Dateien, die für den Betrieb des Online-Compilers benötigt werden. Dazu gehören:

- die Benutzerdaten:
  das Verzeichnis `users`

- die Programmier-Bibliotheken die die Nutzer des Online-Compilers verwenden können:
  das Verzeichnis `lib`

- die Konfigurationsdatei `server.cfg`

- Dateien in der die Übersetzungen der Benutzeroberfläche enthalten sind:
  `zh.lang` für die chinesische Übersetzung und `en.lang` für die englische Übersetzung.

In der Konfigurationsdatei werden alle Einstellungen des Online-Compilers gespeichert. Die Datei (**Java-Properties-Datei** im XML-Format) wird beim ersten Aufruf des Online-Compilers angelegt. Die Einstellungen werden dabei mit Standardwerten initialisiert. Wenn man eine Einstellung ändert, muss der Tomcat-Server neu gestartet werden, damit die Änderung wirksam wird.

Die Einstellungen des Online-Compiler in der Datei `server.cfg` im einzelnen:

Globale Einstellungen des Servlets:

Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
server.log.debug	aktiviert oder deaktiviert das Aufzeichnen von Fehlermeldung zwecks Debugging des Online-Compilers (sollte immer deaktiviert sein)	false	true oder false
server.log.error	aktiviert oder deaktiviert das Aufzeichnen von Fehlermeldungen	true	true oder false
server.log.normal	aktiviert oder deaktiviert das Aufzeichnen von Statusmeldungen	false	true oder false
server.users.guests	gibt an, ob Gast-Anmeldungen erlaubt sind (falls Wert = true)	false	true oder false
server.users.maxfiles	gibt an, wieviel Dateien	50	positive ganze Zahl

Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
	und Verzeichnisse ein Nutzer erstellen darf		
server.users.max	gibt die maximale Anzahl an Nutzern an, die gleichzeitig angemeldet sein dürfen	100	positive ganze Zahl
server.users.maxinactivetime	gibt an, nach welcher Zeit (Sekunden) der Inaktivität eine Client-Verbindung getrennt wird	3600	positive ganze Zahl
server.users.admin.password	das Passwort für den Administrator	„admin“	Text
server.users.admin.name	der Name des Administrators	„admin“	Text
server.examples.importfoldername	der Name des Verzeichnisses, in das Beispiele und ggf. Übungsdateien kopiert werden	„Beispiele + Übungen“	gültiger Verzeichnisname (Unterverzeichnisse getrennt durch „/“)
server.directories.users	Unterverzeichnisname für Nutzer-Dateien	„users“	gültiger Verzeichnisname
server.directories.lib	Unterverzeichnisname für Programmierbibliotheken	„lib“	gültiger Verzeichnisname
server.files.maxsize	maximale Größe für Dateien (in Bytes)	32000	positive ganze Zahl
server.files.maxvariables	maximale Anzahl an Debugger-Variablen pro Datei	20	positive ganze Zahl
server.files.maxtargets	maximale Anzahl an Compiler- und Linker-Targets pro Datei	20	positive ganze Zahl
server.files.maxbreakpoints	maximale Anzahl an Haltepunkten pro Datei	20	positive ganze Zahl
server.session.max	maximale Anzahl an Client-Verbindungen	5000	positive ganze Zahl

Einstellungen für die Benutzer-Authentifizierung mit Ilias:

Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
ilias.server.url	URL der SOAP-Schnittstelle des verwendeten Ilias-Servers	„https://ilias.fh-trier.de/webservice/soap/server.php“	gültige URL
ilias.server.client	Name des Ilias-Clients, der zur Nutzer-Authentifizierung verwendet werden soll	„charly“	Text

Einstellungen für Java:

Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
java.runtime.name	Name des Backends für Java-Programme	„sun“	sun = Java von SUN
java.runtime.processtimeout	Maximale Anzahl Millisekunden, die ein Java-Programm laufen darf	120000	positive ganze Zahl
java.runtime.security	Name der Funktion, die zum Schutz vor fehlerhaften oder schadhaften Nutzer-Programmen verwendet werden soll	„java“	java = Java Security Manager
java.enabled	aktiviert oder deaktiviert Java	true	true oder false

Einstellungen für C und C++:

Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
ccpp.runtime.security	Name der Funktion, die zum Schutz vor fehlerhaften oder schadhaften Nutzer-Programmen verwendet werden soll	„apparmor“	apparmor = Linux AppArmor
ccpp.runtime.processtimeout	Maximale Anzahl Millisekunden, die ein C- oder C++-Programm laufen darf	12000	positive ganze Zahl
ccpp.runtime.name	Name des Backends für C- und C++-Programme	„gnu“	gnu = GNU Compiler Collection
ccpp.enabled	aktiviert oder deaktiviert C und C++	true	true oder false

Einstellungen für Lisp (werden zur Zeit noch nicht verwendet):

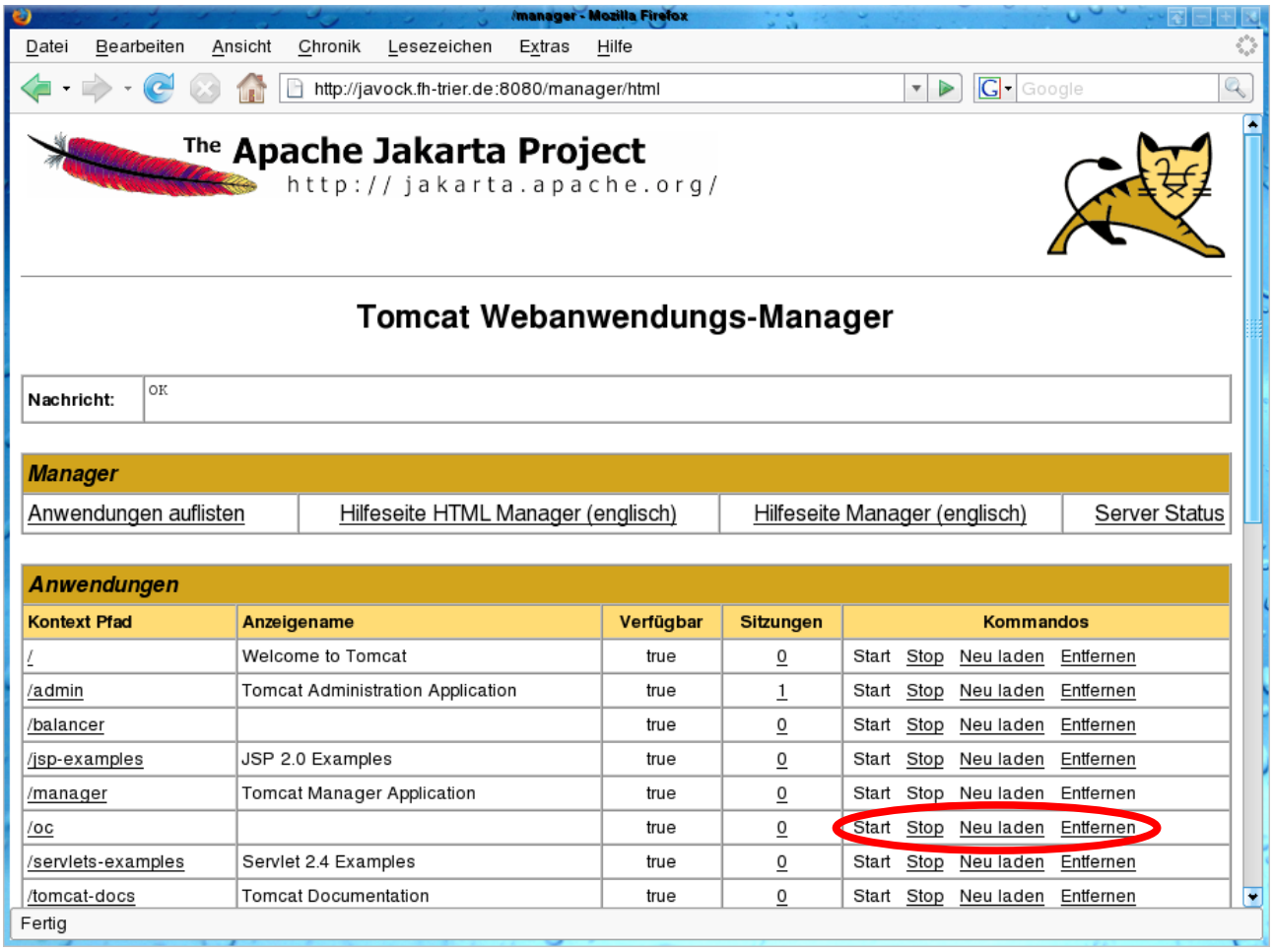
Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
lisp.runtime.name	Name des Backends für Lisp-Programme	„gnu“	gnu = GNU Common Lisp
lisp.enabled	aktiviert oder deaktiviert Lisp	true	true oder false

Einstellungen für C# (werden zur Zeit noch nicht verwendet):

Name	*Beschreibung*	*Standardwert*	*mögliche Werte*
csharp.runtime.name	Name des Backends für C#-Programme	„mono“	mono = MONO C#
csharp.enabled	aktiviert oder deaktiviert C#	true	true oder false

9.2.3 Neustart des Online-Compilers

Jede Änderung der Konfiguration erfordert ein Neustart des Tomcat-Servers oder zumindest ein Neustart des Servlets. Um den Tomcat-Server neu zu starten muss dieser zunächst mit dem Befehl `/usr/share/tomcat5/bin/shutdown.sh` gestoppt, und danach mit dem Befehl `/usr/share/tomcat5/bin/startup.sh` neu gestartet werden.
In der Regel genügt es allerdings, nur das Servlet neu zu laden. Dies kann man am einfachsten über die Manager-Seite des Tomcat-Servers durchführen. Diese Seite erreicht man über den Link „Tomcat Manager“ der Hauptseite des Tomcat-Servers http://javock.fh-trier.de:8080: (Name des Administrators ist „root“ mit dem Passwort „apachetomcatfh“):


/manager - Mozilla Firefox
Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe
http://javock.fh-trier.de:8080/manager/html
The Apache Jakarta Project
http://jakarta.apache.org/
Tomcat Webanwendungs-Manager
Nachricht: OK
Manager
Anwendungen auflisten | Hilfeseite HTML Manager (englisch) | Hilfeseite Manager (englisch) | Server Status
Anwendungen
Kontext Pfad | Anzeigename | Verfügbar | Sitzungen | Kommandos
/ | Welcome to Tomcat | true | 0 | Start Stop Neu laden Entfernen
/admin | Tomcat Administration Application | true | 1 | Start Stop Neu laden Entfernen
/balancer | | true | 0 | Start Stop Neu laden Entfernen
/jsp-examples | JSP 2.0 Examples | true | 0 | Start Stop Neu laden Entfernen
/manager | Tomcat Manager Application | true | 0 | Start Stop Neu laden Entfernen
/oc | | true | 0 | Start Stop Neu laden Entfernen
/servlets-examples | Servlet 2.4 Examples | true | 0 | Start Stop Neu laden Entfernen
/tomcat-docs | Tomcat Documentation | true | 0 | Start Stop Neu laden Entfernen
Fertig
Abbildung 42: Tomcat Manager-Seite

Ein Klick auf „Neu laden“ (rote Markierung) in der Zeile „oc“ lädt das Online-Compiler-Servlet neu.

9.3 Benutzerhandbuch

Dieses Kapitel befasst sich mit der Bedienung des Online-Compilers aus Anwendersicht bzw. mit der Bedienung des Client-Programms (Webinterface oder Benutzerschnittstelle) des Online-Compilers. Im Online-Compiler selbst befindet sich ein ähnliches Dokument unter dem Menüpunkt „HILFE“.

9.3.1 Voraussetzungen

Voraussetzung für den Online-Compiler ist ein aktueller Browser mit aktiviertem **JavaScript** und aktivierten **Cookies**. Getestet wurde der Online-Compiler mit folgenden Browsern:

- Internet Explorer 7.0
- Firefox 2.0

Andere Browser werden nicht unterstützt. Man kann den Online-Compiler aber auch (auf eigene Verantwortung) mit anderen Web-Browsern aufrufen. In die URL des Online-Compilers muss dann der Parameter **browsercheck=no** hinzugefügt werden:

http://javock.fh-trier.de:8080/oc/index.html?browsercheck=no

9.3.2 Das Hauptfenster

Wenn man in der Adresszeile seines Browsers die URL des Online-Compilers (OC) http://javock.fh-trier.de:8080/oc/index.html eingibt oder den Online-Compiler über einen entsprechenden Link in Ilias (http://ilias.fh-trier.de) auswählt, erscheint das Hauptfenster des OC:

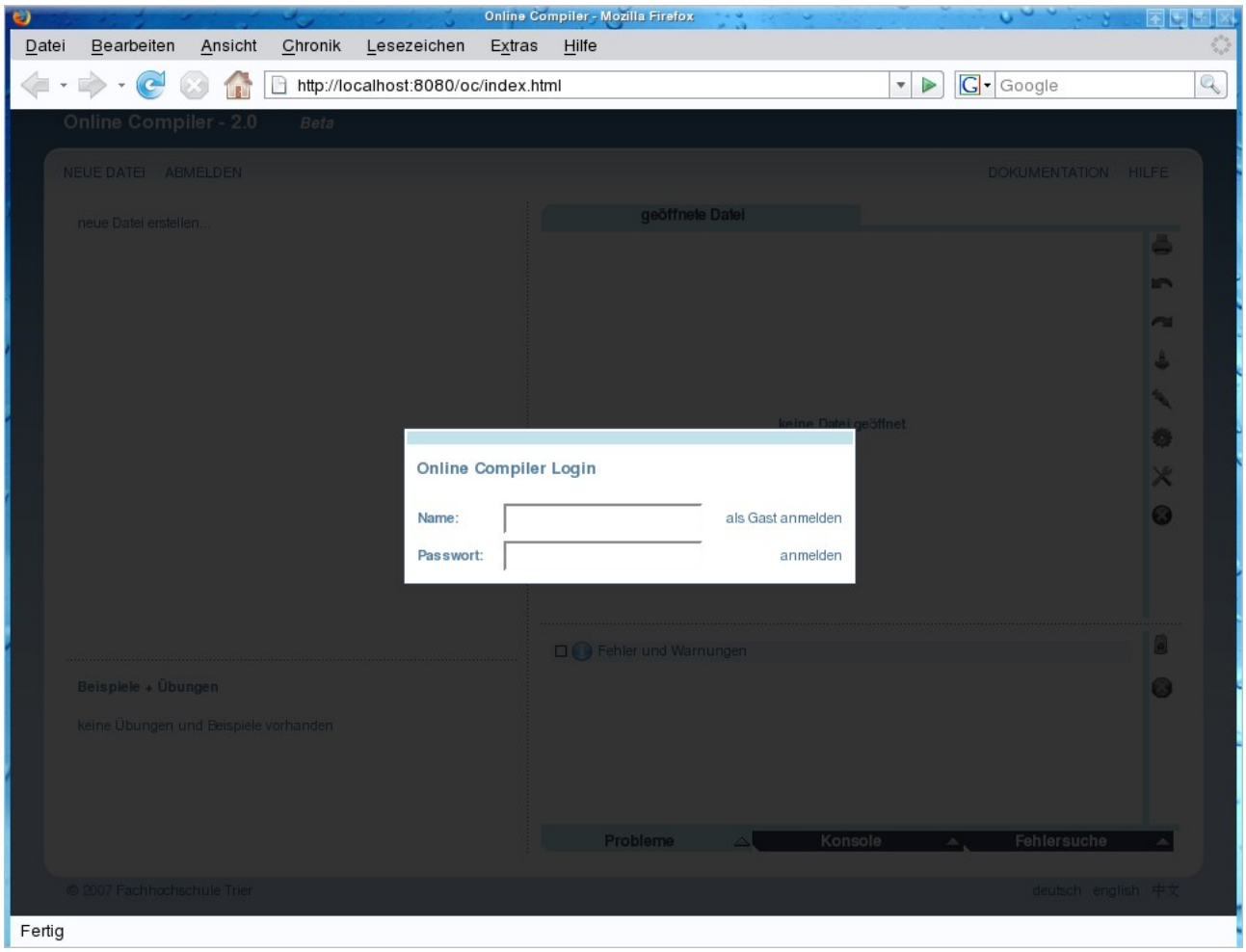


Abbildung 43: Anmeldung-Dialog beim ersten Aufruf des Online-Compilers

Wenn man zum ersten Mal den Online-Compiler aufruft, muss man sich anmelden. Dazu kann man entweder seinen Ilias-Benutzernamen mit dem dazugehörigen Passwort eingeben, oder sich als Gast („als Gast anmelden“) anmelden. Wenn man sich als Gast anmeldet, werden die Dateien, die man erstellt hat, nach dem Abmelden wieder gelöscht.

Nachdem man sich angemeldet hat, kann man auf die Webseite zugreifen:

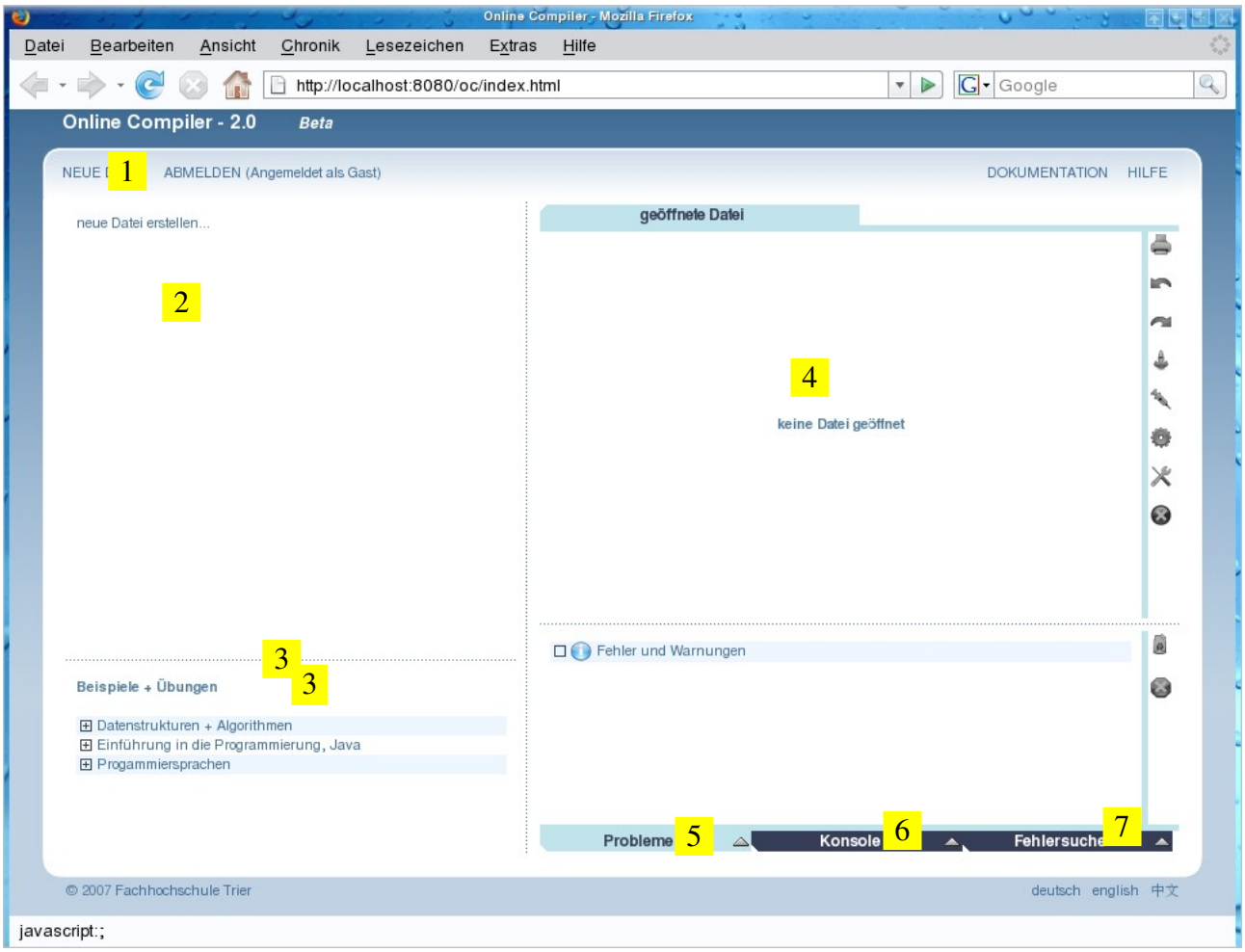


Abbildung 44: Online-Compiler nach der Anmeldung

Bestandteile der Webseite:

1: Menü (siehe nächsten Abschnitt)
2: Falls Dateien oder Verzeichnisse erstellt wurden, werden sie hier angezeigt (Datei-Baum).
3: Liste aller Programm-Beispiele
4: Wenn eine Datei geöffnet wird, kann man hier den Dateiinhalt ansehen und bearbeiten (Editor).
5: Ein Klick auf diesen Tab zeigt Fehlermeldungen und Warnungen, zum Beispiel vom Compiler, an.
6: Ein Klick auf diesen Tab zeigt Ein- und Ausgabe von laufenden Programmen an.
7: Zeigt Informationen zu einem Debugger an. Zum Beispiel Debugger-Variablen.

9.3.3 Menüpunkte und Funktionen

Ein kurze Erläuterung zu den Menüpunkten im Hauptfenster des Online-Compilers:

- NEUE DATEI: Mit diesem Menü kann man eine neue Datei oder ein neues Verzeichnis erstellen. Außerdem kann man eine lokale Datei auswählen und zum Server hochladen.

- ABMELDEN: Meldet den angemeldeten Online-Compiler-Nutzer ab. Danach wird ein Dialog-Fenster geöffnet, mit dem man sich erneut anmelden kann.

  Achtung:
  Wenn man sich als Gast abmeldet, werden alle Dateien gelöscht! Sie sollten deshalb vorher lokal gesichert werden. Wenn man sich als normaler Anwender abmeldet, gehen die Dateien hingegen nicht verloren.

- DOKUMENTATION: Menü über das man Dokumentationen der verschiedenen Programmiersprachen öffnen und durchsuchen kann.

- HILFE: öffnet eine kurze Hilfe für den Online-Compiler

Am unteren Fensterrand kann man die **Sprache** der Benutzerschnittstelle einstellen:

- deutsch: Deutsche Version des Online-Compilers
- english: Englische Version des Online-Compilers
- 中文： Chinesische Version des Online-Compilers

9.3.4 Dateien und Verzeichnisse

Wie im ersten Teil des Handbuches beschrieben wurde, werden im linken Teil des Online-Compiler-Fensters die erstellten Dateien und Verzeichnisse angezeigt (Datei-Baum). Neben jedem Verzeichnis und jeder Datei werden Symbole angezeigt, mit denen diese bearbeitet werden können:

- ein Klick auf dieses Symbol öffnet ein Popup-Menü, mit dem eine neue Datei in einem Verzeichnis erstellt werden kann. Wenn man neue Dateien oder Verzeichnisse im Wurzel-Verzeichnis erstellen möchte, muss man dazu das Menü „NEUE DATEI“ verwenden.
- mit diesem Symbol wird eine Datei oder ein Verzeichnis gelöscht
- mit diesem Symbol kann man eine oder mehrere Dateien compilieren. Wird das Symbol bei einem Verzeichnis angeklickt, werden alle Dateien innerhalb dieses Verzeichnisses compiliert (siehe folgende Abschnitte).
- ein Klick auf dieses Symbol startet ein Programm. Dazu muss die Datei ein gültiges Programm enthalten (in der Regel eine „main“-Funktion). Wenn die Datei noch nicht compiliert wurde, wird sie vor dem Start automatisch compiliert
- ein Klick auf dieses Symbol startet den Debugger mit einem Programm. Dazu muss die Datei ein gültiges Programm enthalten (in der Regel eine „main“-Funktion). Wenn die Datei noch nicht compiliert wurde, wird sie vor dem Start automatisch compiliert
- mit diesem Symbol kann man Dateien oder Verzeichnisse umbenennen
- dieses Symbol dient zum Herunterladen eines Verzeichnisses auf den eigenen Computer. Damit kann man seine Dateien speichern und später wieder auf den Server laden (siehe Kapitel „Dateien lokal speichern“)
- Öffnet ein Programm-Beispiel (siehe folgendes Kapitel)
- Öffnet die Webseite eines Programm-Beispiels im E-Learning-System

Dateien, die gerade geöffnet sind, werden **fett** dargestellt!

9.3.5 Programm-Beispiele

Im Online-Compiler werden alle Programm-Beispiele, die es in den Ilias-Kursen gibt, angezeigt:

Abbildung 45: Programm-Beispiele im Online-Compiler

Die Programm-Beispiele können über Ilias durch einen Klick auf den entsprechenden Link geöffnet werden. Im Online-Compiler selbst können Programm-Beispiele durch einen Klick auf geöffnet werden. Wird ein Programm-Beispiel geöffnet, werden alle Dateien, welche Bestandteil des Programm-Beispiels sind, im Online-Compiler geladen.

Wenn man die Seite eines Beispiels im Ilias-System öffnen möchte, muss man auf klicken.

9.3.6 Dateien bearbeiten

Wenn man auf einen Dateinamen klickt, wird der Dateiinhalt im Editor angezeigt und kann bearbeitet werden:

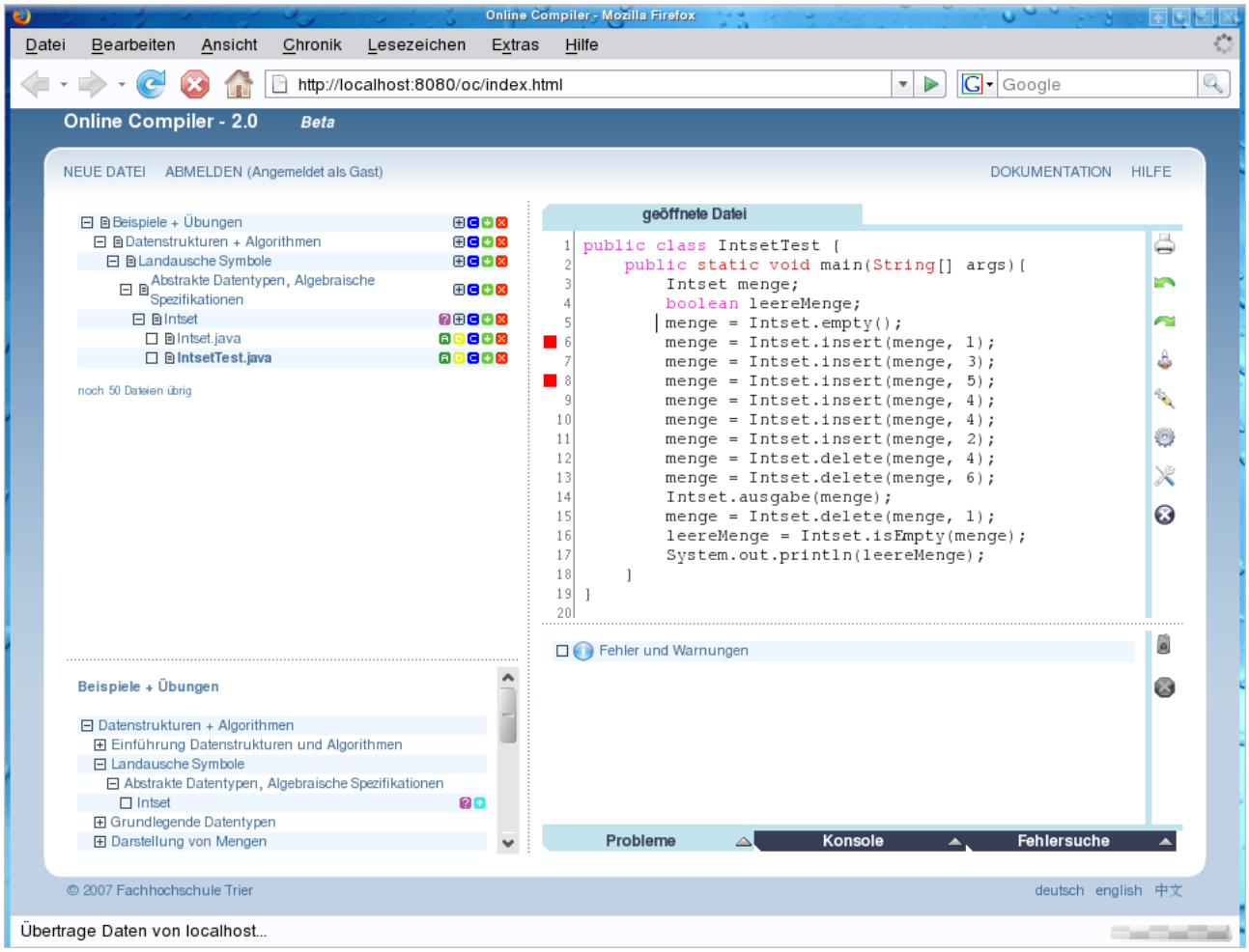


Abbildung 46: geöffnete Datei im Online-Compiler

Wenn man auf eine Zeilennummer klickt, wird ein Debugger-Haltepunkt in der Zeile hinzugefügt. Dass heißt, der Debugger wird beim nächsten Ausführen der Datei in dieser Zeile stoppen. Durch einen weiteren Klick auf den Debugger-Haltepunkt, wird dieser wieder gelöscht.

Schalter des Editors:

- Druckt den Dateiinhalt
- wiederholt die letzte Aktion
- macht die letzte Aktion rückgängig

- Compiliert die Datei (siehe Abschnitt „Compilieren von Dateien“)
- „debuggt“ die Datei (siehe Abschnitt „Programme debuggen“)
- führt ein Programm aus (siehe Abschnitt „Programme ausführen“)
- öffnet ein Popup-Menü zum Konfigurieren der Datei (siehe unten)
- Speichert und Schließt die Datei

Durch einen Klick auf öffnet sich ein Popup-Menü zur Konfiguration der geöffneten Datei. Zur Konfiguration einer Datei zählen die Kommandozeilenargumente und die sogenannten Compiler-Targets. Kommandozeilenargumente (siehe unten) sind die Argumente, die dem Programm beim Start (Ausführen) übergeben werden. Compiler-Targets (siehe unten) sind Dateien, die beim Compilieren einer Datei mit compiliert und ggf. gelinkt werden.

Zum Bearbeiten von Compiler-Targets öffnet sich ein Dialog, in dem die Targets eingegeben werden können. Um eine Datei als Compiler-Target hinzuzufügen, muss einfach der Dateiname per Drag & Drop (mit der Maus) in den Dialog gezogen werden. Um ein Target wieder zu entfernen, muss man einfach auf klicken. Damit eine Datei außerdem gelinkt wird (Linker), muss man nur auf klicken.

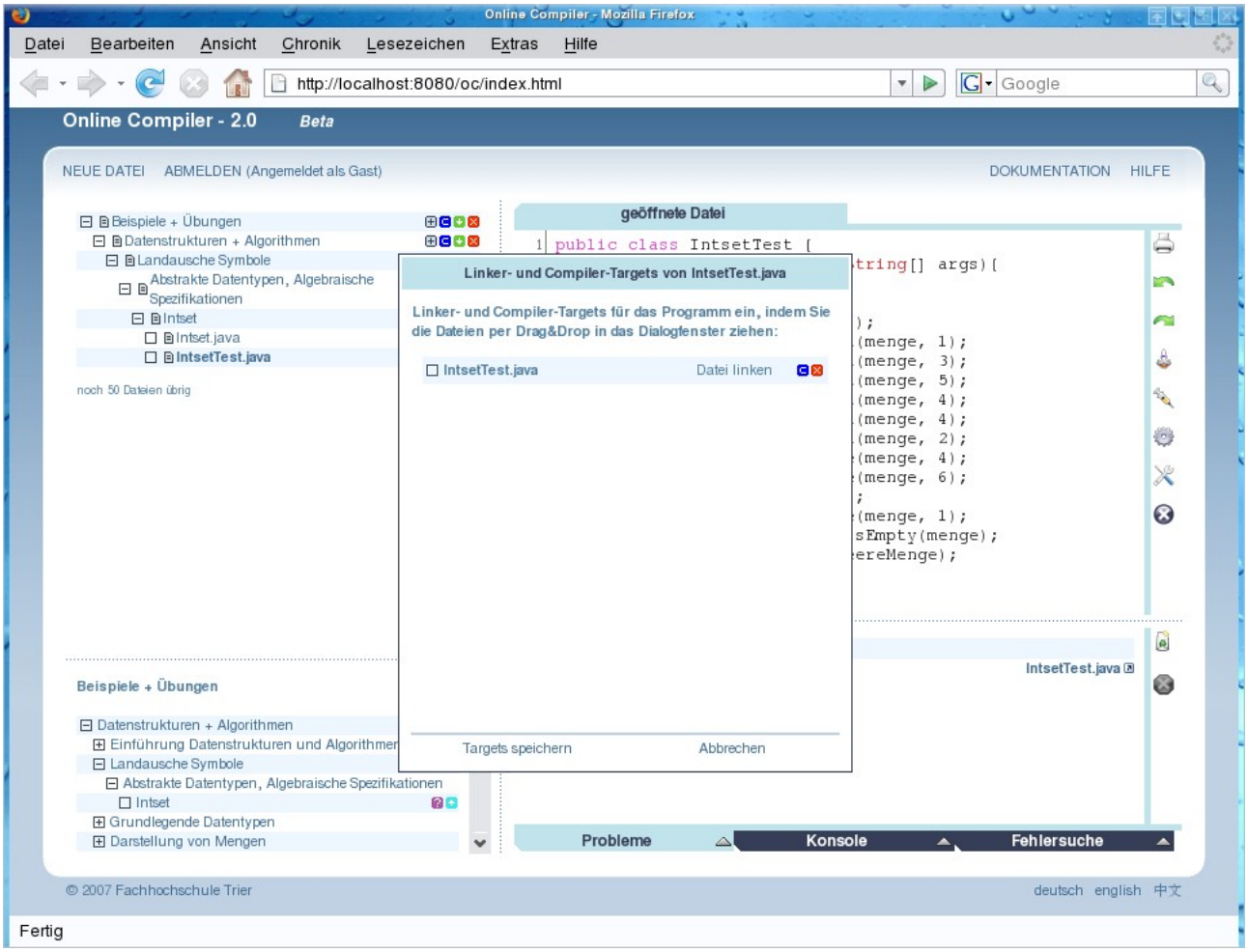


Abbildung 47: Dialog zum Bearbeiten der Targets einer Datei

Zum Bearbeiten von Programm-Kommandozeilenargumenten öffnet sich ein Dialog, in dem die Argumente eingegeben werden können. Die verschiedenen Kommandozeilenargumente müssen durch Leerzeichen voneinander getrennt werden. Einzelne Kommandozeilenargumente können auch in Anführungsstrichen „“ geschrieben werden.

Beispiele für Kommandozeilenargumente:

- „Ein Argument mit Leerzeichen1“ „Ein Argument mit Leerzeichen2“
- ArgumentOhneLeerzeichen1 ArgumentOhneLeerzeichen2 ArgumentOhneLeerzeichen3

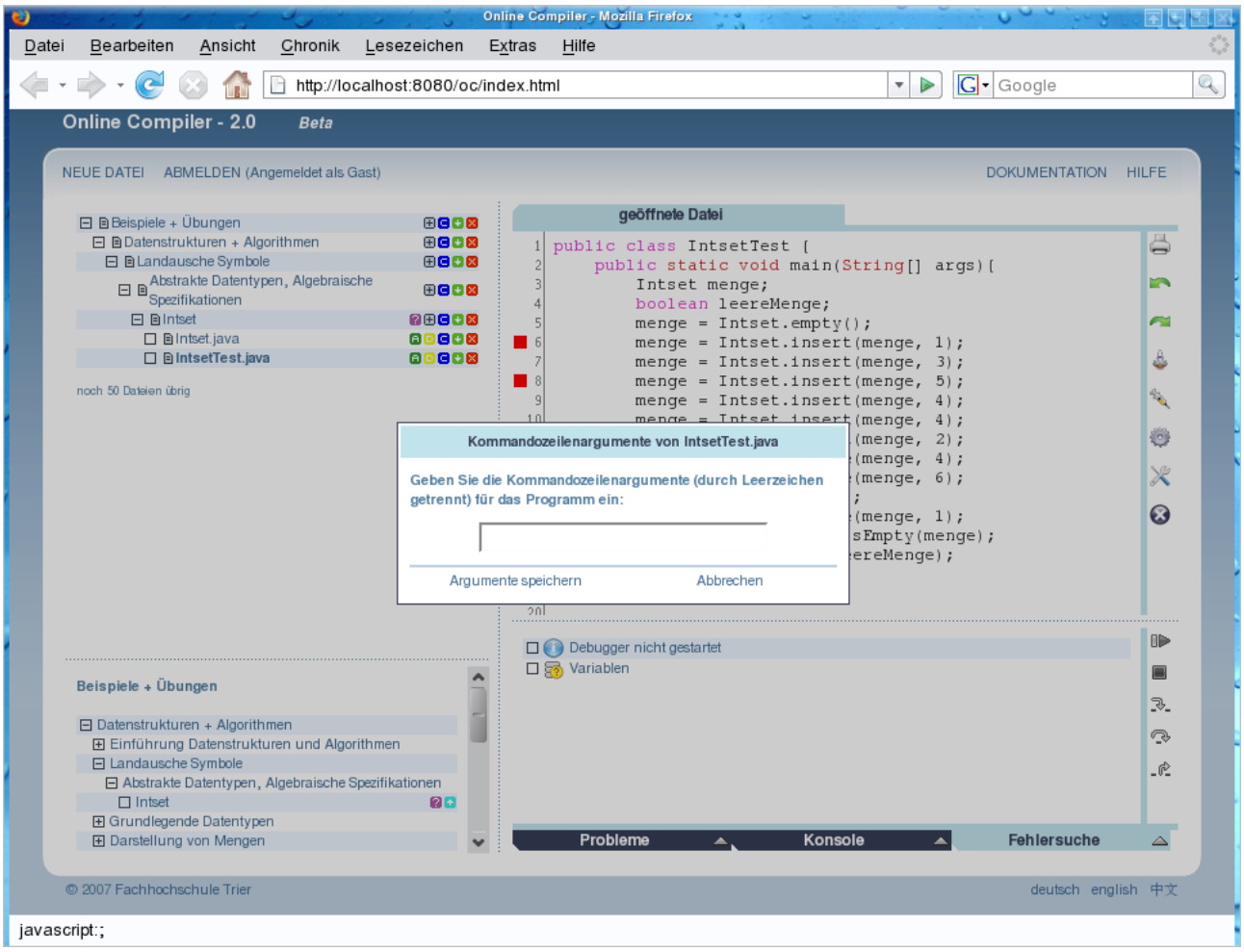


Abbildung 48: Dialog zum Bearbeiten der Kommandozeileargumente einer Datei

9.3.7 Dateien lokal sichern

Alle Dateien, die man im Online-Compiler erstellt hat, können auf dem eigenen Rechner gesichert werden (Download). Dazu muss man auf das Download-Symbol klicken. Daraufhin wird ein Popup-Menü angezeigt, in dem man die Art der Datei, die vom Server heruntergeladen werden soll, angeben kann. Man hat die Wahl zwischen:

- **XML-Datei (XOC):**

  Verzeichnisse und Dateien können als XML-Dateien heruntergeladen werden. Der Vorteil dieses Formates ist, dass alle Informationen der Dateien erhalten bleiben. So werden in den XML-Dateien Debugger-Haltepunkte, Debugger-Variablen, Kommandozeilenargumente usw. gespeichert. Wenn man also alle Daten sichern möchte, sollte man das XML-Format wählen.

- **JAR-Archiv:**

  Verzeichnisse können als JAR-Archive gesichert werden. Dabei gehen allerdings Informationen wie Debugger-Haltepunkte und Kommandozeilenargumente der Dateien verloren.

- **Text-Datei:**

  Quelltextdateien können als normale Text-Dateien gesichert werden. Dabei gehen allerdings Informationen wie Debugger-Haltepunkte und Kommandozeilenargumente verloren.

Dateien können auch zum Server hochgeladen werden. Dazu muss man im Menü „NEUE DATEI“ den Eintrag „lokale Datei öffnen“ auswählen. Alternativ kann man auch auf das Symbol klicken und dort ebenfalls die Eintrag „lokale Datei öffnen“ auswählen. Daraufhin öffnet sich ein Dialog, mit dem man eine lokale Datei auswählen und zum Server hochladen kann. Zum Server können alle Dateitypen hochgeladen werden. Wenn man eine XML-Datei, die man vorher vom Server heruntergeladen hat, zum Server hochlädt, werden die heruntergeladen Dateien komplett wiederhergestellt.

9.3.8 Compilieren von Dateien

Dateien können Compiliert werden. Dazu muss man auf das Compilieren-Symbol in dem Dateibaum klicken. Wenn man eine Datei geöffnet hat, kann diese durch eine Klick auf compiliert werden. Wenn man im Dateibaum auf neben einem Verzeichnisnamen klickt, wird nicht das Verzeichnis „compiliert“, sondern alle Dateien in dem Verzeichnis und dessen Unterverzeichnissen.

Wenn eine Datei compiliert wird, wird automatisch die Registerkarte „Probleme“ angezeigt. In dieser werden alle Fehlermeldungen und Warnungen, aber auch Statusmeldungen, angezeigt. Tritt während der Compilierung einer Datei zum Beispiel ein Fehler auf, wird eine entsprechende Meldung in der Registerkarte dargestellt. Man kann dann, durch einen Klick auf den Dateinamen, die Datei, in der der Fehler aufgetreten ist, öffnen.

Abbildung 49: Registerkarte „Probleme“ zum Anzeigen von Fehlermeldungen

Wie bereits im Abschnitt „Datei bearbeiten“ erwähnt, hat jede Datei ein oder mehrere sogenannte Targets. Targets sind Dateien, die bei einem Klick auf oder compiliert werden. Wenn man also mehrere Targets für eine Datei hat, werden all diese Dateien compiliert. Wenn man hingegen gar kein Target hat, wird auch keine Datei compiliert

C- und C++-Programme müssen nicht nur compiliert, sondern müssen auch mit einem Linker verarbeitet werden. Ein Linker (**Binder**) bindet mehrere compilierte Dateien zu einem ausführbaren Programm zusammen. Dazu werden ebenfalls die Targets verwendet. Für jedes Target kann man festlegen, ob es zu dem Programm gelinkt werden soll oder nicht. Wenn man also bei einer C- oder C++-Datei keine Linker-Datei angegeben hat, wird auch kein ausführbares Programm erzeugt.

9.3.9 Programme ausführen

Mit dem Online-Compiler können Java-, C- und C++-Programme ausgeführt werden. Zum Starten eines Programmes reicht es aus, auf das Symbol zu klicken. Man kann aber auch eine Datei, welches das auszuführende Programm enthält, öffnen und dann auf klicken. Falls das Programm noch nicht compiliert wurde, wird das vor dem Start des Programmes automatisch durchgeführt.

Abbildung 50: Registerkarte „Konsole“ zum Anzeigen von Ein- und Ausgabe von Programmen

Wenn ein Programm gestartet wurde, wird automatisch die Registerkarte „Konsole“ geöffnet, in der die Ausgabe des Programmes angezeigt wird. In dem Eingabefeld „Eingabe“ kann man Text eingeben, der nach einem Klick auf an das Programm gesendet wird.

Weitere Schalter beim Ausführen eines Programmes:

- Löscht die Ausgabe des Programmes
- Stoppt die Ausführung eines Programmes

9.3.10 Programme debuggen

Mit dem Online-Compiler kann man Java-, C- und C++-Programme „debuggen“ beziehungsweise mit einem Debugger ausführen. Um den Debugger zu starten, kann man entweder auf das Symbol klicken oder eine Datei, welche das zu debuggende Programm enthält, öffnen und dann auf das Symbol klicken. Wenn der Debugger gestartet ist, wird automatisch die Registerkarte „Fehlersuche“ angezeigt.

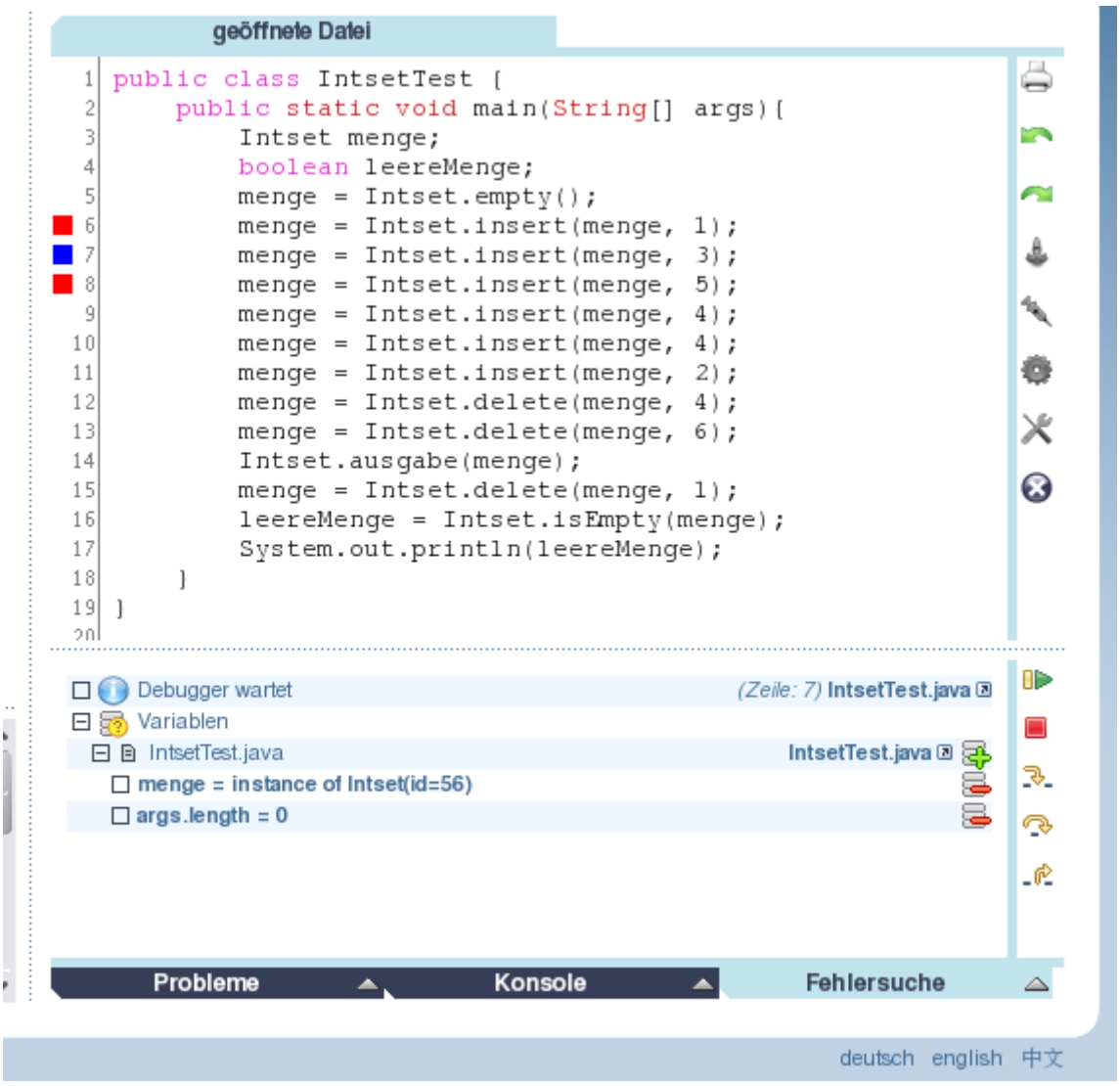


Abbildung 51: Debugger-Ansicht im Online-Compiler

Wenn das zu debuggende Programm an einen Debugger-Haltepunkt angekommen ist, wird das Programm gestoppt. Man hat dann die Möglichkeit, Befehle auszuführen in dem man auf eines der folgenden Symbole klickt:

- Setzt die Ausführung des Programmes fort

- Beendet das Programm und damit auch den Debugger
- Der Debugger springt in eine Funktion (Step-Into)
- Der Debugger springt aus der aktuellen Funktion (Step-Out)
- Führt genau einen Befehl im Programm aus (Step-Over)
- Fügt eine Debugger-Variable hinzu. Eine Debugger-Variable ist eine Variable in dem zu debuggenden Programm. Der aktuelle Wert der Variablen wird in der Registerkarte „Fehlersuche“ angezeigt. Wenn an der aktuellen Position im Programm keine Variable mit dem Namen existiert, wird als Wert „???“ angezeigt.
- Entfernt eine Debugger-Variable

In der Registerkarte „Fehlersuche“ wird neben den **Debugger-Variablen** auch die aktuelle Debugger-Position angezeigt. Also die Position, in der sich das zu debuggende Programm gerade befindet. Dazu wird neben dem Symbol die aktuelle Datei und die aktuelle Zeile des Debuggers angezeigt.

Außerdem wird jedesmal, wenn das zu debuggende Programm stoppt (zum Beispiel weil ein Haltepunkt erreicht wurde) die Datei, in der sich der Debugger gerade befindet, im Editor geöffnet und zur aktuellen Zeile gescrollt. Man hat dann die Möglichkeit, Haltepunkte zu ändern, bevor man den Debugger wieder startet.

9.3.11 Einschränkungen

Da mit dem Online-Compiler Programme auf einem Server ausführt werden können, unterliegen diese Programme einigen Einschränkungen:

- sie dürfen nicht unbegrenzt lange laufen (maximal 2 Minuten)
- einige Funktionen wie Sockets sind gesperrt
- Swing, AWT und andere Toolkits werden nicht unterstützt
- Erstellen von Dateien ist nicht erlaubt
- Schreiben in Dateien ist nicht erlaubt
- Das Aufrufen von anderen Programmen ist verboten
- man darf nur in seinem eigenen Verzeichnis Dateien lesen

Ansonsten kann man im Online-Compiler alle Funktionen der jeweiligen Programmiersprachen nutzen. Ausnahmen sind zur Zeit nur Lisp und C#. Diese beiden Programmiersprachen werden noch nicht vollständig vom Online-Compiler unterstützt. Dass heißt, Lisp- und C#-Dateien können nicht compiliert und ausgeführt werden.

10 Referenzen

[HV 2006] Lars Herwartz, Henning Voß: Erstellen, Test und Integration eines Java-Online-Compilers in einen Java-Online-Kurs
Bachelor Projektarbeit, 2006
Fachhochschule Trier

[SW 2005] Jochen Sonntag, Markus Welter: Erstellung des Online-Kurses „Einführung in die Programmierung mit Java“ mit Hilfe der Plattform WebCT
Bachelor Projektarbeit, 2005
Fachhochschule Trier

[Ilias] Matthias Kunkel: Ilias Open-Source
http://www.ilias.de, 2007,
Download am 18.02.2007

[Sun 1] Sun Microsystems, Inc.: Java Technology
http://java.sun.com, 2007,
Download am 18.02.2007

[Sun 2] Sun Microsystems, Inc.: Java Debug Interface, API Specification
http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi, 2007,
Download am 18.02.2007

[Sun 3] Sun Microsystems, Inc.: Java Platform Debugger Architecture, JDB
http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdb.html, 2007,
Download am 18.02.2007

[Sun 4] Sun Microsystems, Inc.: Java 2 Platform Standard Edition 5.0, API Specification
http://java.sun.com/j2se/1.5.0/docs/api, 2007,
Download am 18.02.2007

[Sun 5] Sun Microsystems, Inc.: Remote Method Invocation
http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, 2007,
Download am 18.02.2007

[FSF 1] Free Software Foundation, FSF: GCC: The GNU Compiler Collection
http://gcc.gnu.org/, 2007,
Download am 18.02.2007

[FSF 2] Free Software Foundation, FSF: GDB: The GNU Project Debugger
http://sourceware.org/gdb/, 2007,
Download am 18.02.2007

[JDT] Eclipse Foundation, Inc.: Eclipse Java Development Tools
http://www.eclipse.org/jdt/, 2007,
Download am 18.02.2007

[Novell 1] Novell, Inc.: AppArmor Application Security for Linux
http://www.novell.com/linux/security/apparmor/, 2007,
Download am 20.02.2007

[Novell 2] Novell, Inc.: .NET Mono Project
http://www.mono-project.com/Main_Page, 2007,
Download am 20.02.2007

[Apache 1] The Apache Software Foundation: Apache Tomcat
http://tomcat.apache.org/, 2007,
Download am 10.02.2007

[Apache 2] The Apache Software Foundation: Apache HTTP Server Project
http://httpd.apache.org/, 2007,
Download am 22.02.2007

[BSI Tomcat 1] Bundesamt für Sicherheit in der Informationstechnik, BSI:
Sicherheitsuntersuchung des Apache Jakarta Tomcat Servlet
Containers
http://www.bsi.bund.de/literat/studien/tomcat/feinkonzept.pdf, 2007,
Download am 10.02.2007

[Koch 2006] Peter-Paul Koch: Benchmark - W3C DOM vs. innerHTML
http://www.quirksmode.org/dom/innerhtml.html, 2007,
Download am 16.02.2007

[Mozilla 1] Mozilla Foundation: MIDAS Specification
http://www.mozilla.org/editor/midas-spec.html, 2007,
Download am 18.02.2007

[Mozilla 2] Mozilla Foundation: Home of the Firefox web browser
http://www.mozilla.com/en-US/, 2007,
Download am 18.02.2007

[Opera 1] Opera Software ASA: Opera Web-Browser
http://www.opera.com/, 2007,
Download am 18.02.2007

[KDE 1] The K Desktop Environment Project: Konqueror Web-Browser
http://www.konqueror.org/, 2007,
Download am 18.02.2007

[Apple 1] Apple, Inc.: Mac OS X Web-Browser Safari
http://www.apple.com/de/macosx/features/safari/, 2007,
Download am 20.02.2007

[Microsoft 1] Microsoft Corporation: Internet Explorer 7 Web-Browser
http://www.microsoft.com/germany/windows/ie/default.mspx, 2007,
Download am 20.02.2007

[Li Shen 2006] Li Shen: ACE JavaScript Component for Ajax
LINK, 2005,
Download am 01.01.2007

[JSXML] Jon van Noort, David Joham: XML for JavaScript
http://xmljs.sourceforge.net/, 2006,
Download am 01.01.2007

[Herold 1999] Helmut Herold: C-Kompaktreferenz
Addison Wesley Longman Verlag, 1999

[Liu 2004] M.L. Liu: Distributed Computing, Principles and Applications
Addison Wesley Verlag, 2004

[Google 1] Google, Inc.: Google Docs
http://docs.google.com, 2007,
Download am 21.02.2007

11 Erklärung

Hiermit erkläre ich, Henning Voß, dass ich die vorliegende Arbeit unter Verwendung der angegebenen Quellen eigenhändig erstellt habe.

Trier, den 22. Februar 2007

_____

Henning Voß